Библиографический список

1. *Якимов А.И.* Имитационное моделирование в ERP-системах управления / А.И. Якимов, С.А. Альховик. – Минск: Бел. наука, 2005. – 197 с.
2. *Якимов А.И.* Модернизация программно-технологического комплекса имитации сложных систем BelSim для организации распределенных вычислений / А. И. Якимов // Информатика. - 2008. - №2.– С. 137-142.
3. *Шпаковский Г.И.* Программирование многопроцессорных систем в стандарте MPI / Г.И. Шпаковский, Н.В. Серикова. – Минск: БГУ, 2002. – 323 с.
4. *Альховик С.А.* Генетический алгоритм в задаче оптимизации плана грузоперевозок / С.А. Альховик, А.В. Сазоненко, А.А. Ковалевич // Известия Гомельского государственного университета им. Ф. Скорины. – 2006. – 4 (37). – С. 110-112.
5. *Якимов А.И.* Автоматизация эксперимента на примере исследования генетического алгоритма / А.И. Якимов, В.В. Башаримов, С.А. Альховик // Автоматизация и современные технологии. - 2006. - №5. – С. 3-9.
6. *Спиридонов А.А.* Планирование эксперимента при исследовании технологических процессов / А.А. Спиридонов. – М.: Машиностроение, 1981. – 184 с.

МЕТОДЫ И СРЕДСТВА ОПТИМИЗАЦИИ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ НА БАЗЕ ЭВОЛЮЦИОННЫХ МОДЕЛЕЙ И АЛГОРИТМОВ

Е.Б. Замятина, С.А. Ермаков [12]

Пермский государственный университет
e_zamyatina@mail.ru,

РЕАЛИЗАЦИЯ РАСПРЕДЕЛЕННЫХ АЛГОРИТМОВ ДЛЯ СИСТЕМЫ ИМИТАЦИОННОГО МОДЕЛИРОВАНИЯ TRIAD.NET

Введение

Имитационное моделирование (ИМ) — это метод, позволяющий строить модели, описывающие процессы так, как они проходили бы в действительности. ИМ является наиболее мощным, а иногда даже единственным способом исследования динамического поведения сложных систем.

Развитие технологий ставит все более сложные задачи перед ИМ, высокие темпы развития делают время определяющим фактором, поэтому вычислительных мощностей одного даже суперкомпьютера не хватает для выполнения моделирования в приемлемые сроки[1,2,3]. Помимо этого, существует еще ряд причин, по которым около 20 лет назад начался переход от последовательного моделирования к распределенному (например, объединение уже готовых систем имитации или тренажеров (на основе технологии HLA – High Level Architecture))[4,5].

Несмотря на то, что данное направление развивается уже сравнительно долго (для компьютерных наук), оно по-прежнему остается важным и перспективным и имеет многочисленные применения в физике, экономике и управлении.

При последовательном выполнении программы симулятор рабо-

[12] Работа выполнена при поддержке гранта РФФИ 08-07-90005-Бел_а

тает в цикле, каждый раз выбирая событие с минимальным штампом локального времени из глобального списка событий и обрабатывая его. Обработка события может привести к изменению состояния системы в целом, планируя некоторое новое число событий для моделирования в будущем, при этом необходимо соблюдать отношения причинности. При последовательном моделировании это обеспечивается автоматически. Параллельное выполнение модели ускоряет выполнение программы, однако необходимо соблюдать причинно-следственные отношения.

Моделируемая система разбивается на подсистемы (логические процессы) каждый из которых последовательно обрабатывает события как при последовательном моделировании. Опишем математическую модель распределенной системы имитационного моделирования: $Model = \bigcup_k lp_k$, где lp_k – это логический процесс; $lp_k = \{G_k, T_k, IN_k OUT_k\}$, где T_k – значение локальных часов, G_k – подграф модели; $IN_k = \{i_1, i_2, i_3, \ldots, i_n\}$ – множество процессов, которые могут послать сообщение lp_k, где i_j – номера процессов; $OUT_k = \{i_1, i_2, i_3, \ldots, i_r\}$ – множество процессов, которым lp_k посылает сообщение; $G_k = \{n_1, n_2, \ldots, n_c\}$ – множество вершин графа модели, где каждая вершина $n_p = \{P_p, R_p, C_p\}$, где $P_p = \{p_1, p_2, \ldots p_r\}$ список полюсов вершины; R_p – рутина наложенная на вершину, $C_p = \{(Q_1^p, t_1^p), (Q_2^p, t_2^p), \ldots, (Q_m^p, t_m^p) \mid t_1^p \le t_2^p \le \ldots \le t_m^p\}$ – локальный календарь событий для вершины, где Q_i^k – событие в списке, а t_i^k – время, на которое запланировано данное событие. События в календаре упорядочены по возрастанию временной метки.

В результате выполнения обработчика события или сообщения, выполняемого рутиной вершины, вершина может послать сообщение другой вершине через 1 или несколько своих полюсов. При этом, вершина-получатель получит сообщение на свой полюс с таким же именем. В список $M_k = \{i_1, i_2, i_3, \ldots, i_n\}$ включаются только те вершины, которые посылают сообщения данной вершине. Общий формат сообщения таким образом выглядит так: $Ms = \{Dn, Dp, t, Message\}$, где Dn – вершина, который мы посылаем сообщения, Dp – полюс через который посылается сообщение. Если он отсутствует, то сообщение передается через все полюса. t – временная метка сообщения (определяется текущим временем процесса-отправителя на момент отправки), Message –содержание сообщения.

Текущее время всей модели в этом случае $TIME = \min\{T_k\}$.

Логический процесс, получив данное сообщение, обрабатывает его с помощью рутины (рутина – подпрограмма, интерпретирующая поведение одного из моделируемых объектов).

Полученное сообщение может содержать время, меньшее локального времени данного процесса. При этом вполне вероятно, что данный логический процесс при обработке сгенерирует событие со штампом времени меньше текущего, а это означает временной парадокс. Временной парадокс опасен тем, что процесс моделирования может привести к другому результату. Для борьбы с парадоксами времени должен быть реализован алгоритм синхронизации модельного времени. Традиционно используются два подхода к реализации алгоритма синхронизации: консервативный и оптимистический [4,6,7,8].

Краткое описание TRIAD.NET

Triad.Net – событийно ориентированная система имитационного моделирования.

Основным способом моделирования в системе является описание модели с помощью языка Triad. Основная его особенность – это разделение модели на слои структур, рутин (поведения) и сообщений. Слой структур описывает структуру соединений элементов вычислительной системы, слой рутин задает принцип их работы, а слой сообщений описывает сигналы, передаваемые между ними. Описание модели может проходить поэтапно, начиная с описания структуры. Все слои описывать не обязательно. Для каждого слоя есть свой круг задач, формулируемых и решаемых в случае его задания. Структура модели может меняться динамически.

Разработчики системы имитации предусмотрели создание компилятора с языка Triad, который строит представление модели (в т.ч. исполняемый код рутин) в объектном виде, т.о. преимуществом системы Triad.Net является то, что построенная пользователем модель не интерпретируется, а выполняется. Таким образом, достигается значительный прирост в эффективности. Такие параметры, как количество прогонов модели, перечень показателей, которые предполагается определить во время имитационного прогона описывают на уровне языка Triad в специальном разделе – условия моделирования. Алгоритмы сбора статистики определяют в информационных процедурах алгоритма анализа модели и в специальной языковой единице – условиях моделирования.

Общее представление распределенной системы TRIAD.Net

Структуру распределенной модели можно представить в виде графа логических процессов (рис. 1).

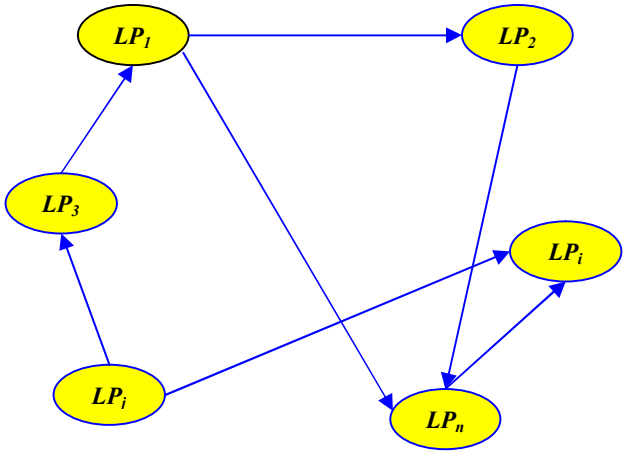

Рис. 1. ***Граф логических процессов***

Прокомментируем рисунок. Направленная дуга между LP_i и LP_j означает, что какая-либо вершина в LP_i передает сообщение вершине в LP_j. Очевидно, что количество связей между логическими процессами должно быть минимально. Это позволит увеличить параллелизм модели.

Рассмотрим структуру каждого логического процесса подробнее (рис.2). На схеме приведена общая структура логического процесса и выделен пример реализации модели в конкретном случае.

Итак, логический процесс состоит из 3 модулей:

- Подсистемы коммуникации, которая реализует взаимодействие между процессами.
- Модуля управляющей программы, реализующей конкретный распределенный алгоритм.
- Подграфа модели – часть графа модели, которая обрабатывается данным логическим процессом.

Модули управляющей программы в конкретной реализации могут использовать дисковое пространство, СУБД, и другие вспомогательные «устройства».

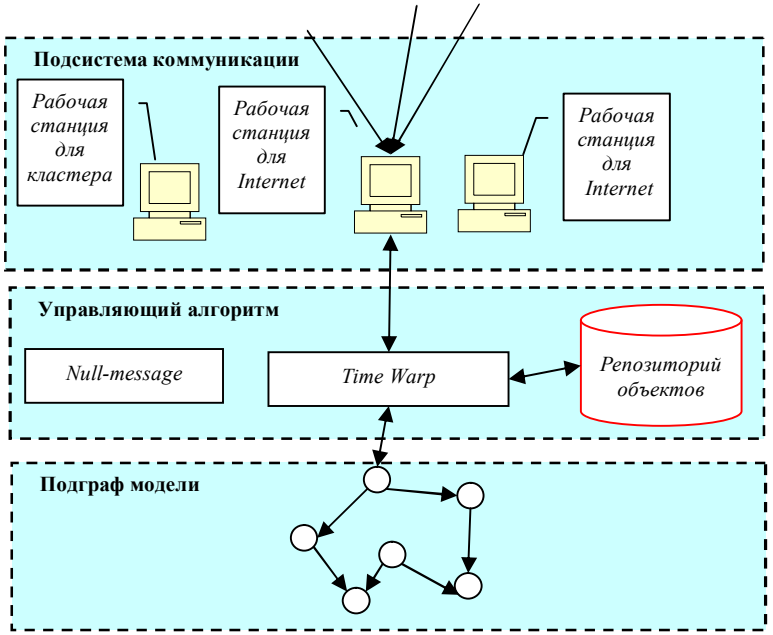

Рис.2. Настройка распределенной системы моделирования(Null-message-консервативный алгоритм с нулевыми сообщениями, Time Warp-оптимистический алгоритм)

Такое разбиение логического процесса на модули обеспечивает гибкость системы, а именно:

- возможность подключения различных модулей подсистемы коммуникации в зависимости от того, в каких условиях реализуется распределенная система: кластер, локальная сеть, глобальная сеть.
- возможность подключения различных модулей управляющей программы в зависимости от эффективности конкретного алгоритма синхронизации, применяемого для конкретной системы.

Консервативный алгоритм

Пусть моделируемая система состоит из нескольких физических процессоров (далее *PP*), каждый физический процесс моделируется отдельным логическим процессом (*LP*). Если в реальной системе PP_i может обмениваться информацией с PP_j то в модели существует канал

$(i; j)$; информацию, передаваемую по каналу $(i; j)$ в момент времени t можно представить как пару из сообщения M и метки времени t. Консервативные алгоритмы синхронизации обеспечивают выполнение следующих правил:

- *Правило монотонности*: пусть $<M1;t1>,<M2;t2>, \ldots, <Mk;t_k>, \ldots$ – последовательность сообщений посланных по каналу $(i; j)$, тогда $t1 \le t2 \le \ldots \le t_k \le \ldots$
- *Правило корректности*: LP_i посылает сообщение LP_j с пометкой времени t, если и только, если PP_i посылает сообщение PP_j в момент времени t.
- *Правило продвижения*: LP_i может продвинуть свои локальные часы (путём обработки сообщения, или выполнения своих процедур) до времени t только в том случае, если $t \le min\ tr_i$, где tr_i – локальное время процесса LP_r, (следует отметить, что в множество LP_r включаются все процессы, которые могут послать сообщение процессу с номером i). В противном случае есть возможность получения "отстающего" сообщения.

Классическим алгоритмом является алгоритм с посылкой нулевых сообщений, разработанный в 1978г. учёными J. Misra и K. M. Chandy. В его основе лежит посылка нулевых сообщений. В отличие от обычных сообщений $<M, t>$, означающих, что в реальной системе было послано сообщение M в момент времени t, сообщение <NULL, t>, посланное по каналу $(i; j)$ означает, что в реальной системе PP_i в момент времени t ничего не посылал PP_j. Таким образом, локальные процессы системы получают информацию о локальном времени других локальных процессов.

Однако такая схема не обеспечивает защиту от возможности возникновения тупика в системах, где возможны циклические передачи сообщений. В системе распределенного имитационного моделирования Triad.Net такие передачи вероятны. Эту проблему можно решить за счет внешней системы, определяющей наличие тупика: каждый локальный процесс, если он находится в ожидании, хранит список тех процессов, которые он ожидает (их может быть несколько). Управляющая система постоянно проверяет указанные списки и определяет наличие тупика (локального или глобального), а затем посылает сообщение одному из локальных процессов о том, что ему можно продвинуть время на минимальную величину.

Другой способ усовершенствования состоит в том, чтобы использовать так называемый *lookahead* – параметр, который определяет горизонт времени, до которого события считаются безопасными. Поскольку модель Triad.Net не содержит дополнительной информации

для вычисления *lookahead*, то его будет задавать пользователь для вершины или группы вершин в языковой единице «условия моделирования». Таким образом, мы немного расширим язык описания модели.

В системе Triad.Net все вершины связаны через полюсы: имя полюса уникально в системе: если у вершин $V1$ и $V2$ есть входной полюс A то они обе синхронно будут получать все сообщения, передаваемые через этот полюс. Алгоритм получения всех внешних входящих каналов сообщений вычисляет все входящие полюсы, принадлежащие вершинам текущего логического процесса, и ищет их в выходящих полюсах всех остальных вершин:

Таким образом, достаточно простым способом каждый логический процесс может определить список внешних полюсов.

Однако наличие внешних полюсов вовсе не означает, что через них будет происходить взаимодействие, поэтому необходимо на этапе трансляции рутин дополнительно определять, на какие полюса рутина будет отправлять сообщения и при наложении рутины на вершину графа отсекать список внешних полюсов вершины.

Оптимистический алгоритм

В отличие от консервативных алгоритмов, не допускающих нарушениё ограничения локальной каузальности, оптимистические методы не следят за этим ограничением. Однако этот подход требует выявления нарушения каузальности и его устранение.

Алгоритм, разработанный Jefferson – Time Wrap

Когда логический процесс получает событие, имеющее временную отметку меньшую, чем уже обработанные события, он выполняет откат и обрабатывает эти события повторно в хронологическом порядке. Откатываясь назад, процесс восстанавливает состояние, которое было до обработки событий (используются контрольные точки) и отказывается от сообщений, отправленных «откаченными» событиями. Известно, что для отката от этих сообщений используется механизм антисообщений (антисообщение – это копия ранее отосланного сообщения). Если антисообщение и соответствующее ему сообщение (позитивное) хранятся в одной и той же очереди, то они взаимно уничтожаются. Чтобы изъять сообщение, процесс должен отправить соответствующее антисообщение. Если соответствующее позитивное сообщение уже обработано, то процесс-получатель откатывается назад. При этом могут появиться дополнительные антисообщения. При ис-

пользовании этой рекурсивной процедуры все ошибочные сообщения будут уничтожены.

Для того чтобы оптимистический алгоритм стал надёжным механизмом синхронизации, необходимо решить 2 проблемы:

Некоторые действия процесса, например, операции ввода-вывода, нельзя «откатить»;

Обсуждаемый алгоритм требует большого количества памяти (для восстановления состояний процессов в контрольных точках, которые создаются независимо от того, произойдёт откат или нет). Требуется особый механизм, чтобы освободить эту память.

Обе эти проблемы решаются с помощью глобального виртуального времени (*GVT*). *GVT* – это нижняя граница временной отметки любого будущего отката. *GVT* вычисляется с учётом откатов, вызванных сообщениями, поступивших «в прошлом». Таким образом, наименьшая временная метка среди необработанных и частично обработанных сообщений и есть *GVT*. После того, как значение *GVT* вычислено, фиксируются операции ввода вывода, выполненные в модельное время, превышающее *GVT*, а память восстанавливается (за исключением одного состояния для каждого из логических процессов).

Вычисление *GVT* очень похоже на вычисление *LBTS* в консервативных алгоритмах. Это происходит потому, что откаты вызваны сообщениями или анти-сообщениями в прошлом логического процесса. Следовательно. *GVT* – это нижняя граница временной метки будущих сообщений (или анти-сообщений), которые могут быть получены позже.

Реализация такого алгоритма наиболее проста, однако требует больших затрат памяти.

В распределенной системе Triad.Net *GVT* реализовано через специальное широковещательное сообщение, которое отправляется сразу всем процессам. В ответ на него локальные процессы присылают свое текущее локальное время *GVT* = минимум локальных времен.

Сохранение состояний подмодели происходит очень быстро благодаря механизму сериализации. Кроме того, все состояния модели хранятся на диске, что не требует затрат оперативной памяти. В оперативной памяти хранится лишь *k* последних состояний, где *k* – параметр, задаваемый пользователем.

При перевычислении *GVT* состояния, имеющие временную метку меньше *GVT*, удаляются. Поскольку частое восстановление состояния системы (особенно каскадное) приводит к значительному замедлению выполнения моделирования, необходимо ограничить стремительное продвижение времени локального процесса. По аналогии с *Lookahead*,

для оптимистического алгоритма пользователь задает параметр *W*, который определяет временное окно [*GVT*, *GVT+W*]. Процесс обрабатывает только те события, временные метки которых находятся в данном временном интервале.

Можно сделать следующий вывод.

Как в консервативном, так и в оптимистическом алгоритмах эффективное выполнение обеспечивается не только тем, насколько быстро выполняется код модели, и насколько быстро функционирует коммуникационная подсистема, а тем, насколько более детальную информацию мы получим о модели. Следует заметить, что структура Triad сама по себе в значительной степени соответствует параллельной модели: каждая вершина имеет свой список событий и обрабатывает только его, связь реализуется лишь через механизм передачи сообщений. Triad – Это мощное средство описания моделей, но он не может обеспечить описание “неформальных” знаний пользователей о моделируемой системе. Таких, например, как: «После 4х вечера в магазине обычно нет посетителей». Как видно такие знания сильно отличаются от формального представления модели, и к тому же вносят нечеткость.

Заключение

Таким образом, перспектива повышения эффективности распределенных алгоритмов состоит в следующем:

- Необходимо разработать язык описания дополнительной информации о модели в терминах пользователей (в виде правил).
- Необходимо добавить в алгоритм моделирование использование этой дополнительной информации. Алгоритм должен уметь определять безопасность события, которое нужно обработать на основе нечетких знаний, представленных пользователем.

Библиографический список

1. *Meyer R.A., Bagrodia R.* Parsec User Manual. Release 1.1., UCLA Parallel Computing Laboratory, 1998. [Электронный ресурс] [Режим доступа: pcl.cs.ucla.edu/projects/parsec].
2. *Premore B.J., Nicol D.M.*. Parallel Simulation of TCP/IP Using TeD // Proceedings of the 1997 Winter Simulation Conference / S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson S., eds. P. 436-447.
3. *Yuan C.*, *Xianlong J.*, *Zhi L.* A distributed simulation system and its application. Simulation Modelling. Practice and Theory. Vol. 15,

Issue 1, January, 2007. Pp.21-31.
4. *Fujimoto R.M.* Distributed Simulation Systems // Proceedings of the 2003 Winter Simulation Conference / S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds. P. 124-134.
5. *Nance R.E.* Distributed Simulation With Federated Models: Expectations, Realizations And Limitations // Proceedings of the 1999 Winter Simulation Conference / P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, eds. P. 1026-1031.
6. *Вознесенская Т.В.* Математическая модель алгоритмов синхронизации времени для распределённого имитационного моделирования // Программные системы и инструменты: Тематический сборник факультета ВМиК МГУ им. Ломоносова, №1. С. 56-66.
7. *Окольнишников В.В.* Представление времени в имитационном моделировании // Вычислительные технологии. Т. 10, №5. Сибирское отделение РАН, 2005. С. 57-77.
8. *Замятина Е.*Б. Материалы специального курса «Современные теории имитационного моделирования». [Электронный ресурс] [http://www.microsoft.com/Rus/Msdnaa/Curricula/Default.mspx].

Е.Б. Замятина, А.Ю. Ефимов, А.А. Козлов[13]

Пермский государственный университет
e_zamyatina@mail.ru

АРХИТЕКТУРА ПОДСИСТЕМЫ МУЛЬТИАГЕНТНОЙ БАЛАНСИРОВКИ В TRIAD.NET

Введение

Высокопроизводительные вычисления завоевывают все более прочные позиции при решении разного рода задач (в том числе задач, в которых применяются методы имитационного моделирования), используя ресурсы нескольких исполнителей для выполнения вычислений. Основная цель использования этих средств – оптимизация времени вычислений. Однако гетерогенность исполнителей (вычислительные узлы имеют разную производительность, линии связи между узлами обладают разной пропускной способностью), гетерогенность самого параллельного приложения (приложение представляет собой совокупность логических процессов, расположенных на разных вычислительных узлах и взаимодействующих посредством посылки сообщений друг другу) приводит к возникновению дисбаланса нагрузки на вычислительных узлах. В результате выигрыш от использования нескольких исполнителей при выполнении вычислений сводится к нулю.

Для того, чтобы избежать нежелательных последствий дисбаланса используют специальное программное обеспечение, реализующее алгоритм балансировки. Алгоритм балансировки предназначен для равномерного распределения нагрузки на вычислительные узлы. Если на каком-нибудь вычислительном узле нагрузка превышает допустимую, то, следуя алгоритму балансировки, часть нагрузки переносят на другой, менее загруженный узел. При этом следует учитывать затраты приложения на коммуникацию между вычислительными узлами.

В настоящей работе в качестве приложения рассматривается распределенная система имитации (Triad.Net) [1, 2] (в дальнейшем будем

[13] Работа выполнена при поддержке гранта РФФИ 08-07-90005-Бел_а