по проблематике ИИ и конструирования интеллектуальных систем, проводимых на кафедрах и в подразделениях МЭИ (ТУ).

В заключение отметим, что ученые и специалисты МЭИ (ТУ), занимающиеся данной проблематикой, активно взаимодействуют с учебными, академическими и научно-исследовательскими институтами России (МГУ им. М.В. Ломоносова, МИФИ, МГТУ им Н.Э. Баумана, МИРЭА, МЭСИ, МИСИС, ТГТУ (г. Тверь), ТТИ ЮФУ (г. Таганрог), ВЦ РАН, ИПУ РАН, ИПС РАН, ИСА РАН, ВИНИТИ РАН, РосНИИ ИТиАП, НПО «АЛЬТАИР», НПО «ЦНИИКА» и другими), а также стран СНГ и дальнего зарубежья.

Литература

1. Московский энергетический институт (технический университет). 1930–2005. М.: Изд-во МЭИ, 2005. 456 с.
2. Институт автоматики и вычислительной техники Московского энергетического института (технического университета) (1958–2008); под. ред. В.П. Лунина, О.С. Колосова. М.: Издат. дом МЭИ, 2008. 256 с.
3. Поспелов Д.А. Логико-лингвистические модели в системах управления. М.: Энергоиздат, 1981. 232 с.
4. Вагин В.Н. Дедукция и обобщение в системах принятия решений. М.: Наука. Глав. ред. Физматлит, 1988. 384 с.
5. Башлыков А.А., Еремеев А.П. Экспертные системы поддержки принятия решений в энергетике; под ред. А.Ф. Дьякова. М.: Изд-во МЭИ, 1994. 216 с.
6. Вагин В.Н., Головина Е.Ю., Загорянская Н.А., Фомина М.Б. Достоверный и правдоподобный вывод в интеллектуальных системах; под ред. В.Н. Вагина, Д.А. Поспелова. М.: Физматлит, 2004. 704 с.
7. Башмаков А.И., Башмаков И.А. Интеллектуальные информационные технологии: учеб. пособие. М.: Изд-во МГТУ им. Н.Э. Баумана, 2005. 312 с.
8. Еремеев А.П. К 75-летию МЭИ. Становление и развитие идей искусственного интеллекта в научной школе Московского энергетического института (технического университета) // Новости искусственного интеллекта. 2005. № 2. С. 63–66.
9. Вагин В.Н., Еремеев А.П. Исследования и разработки кафедры прикладной математики по конструированию интеллектуальных систем поддержки принятия решений на основе нетрадиционных логик // Вест. МЭИ. 2008. № 5. С. 16–26.
10. Вагин В.Н., Еремеев А.П. Некоторые базовые принципы построения интеллектуальных систем поддержки принятия решений реального времени // Изв. РАН. Теория и системы управления. 2001. № 6. С. 114–123.
11. Vagin V.N., Yeremeyev A.P. Modeling Human Reasoning in Intelligent Decision Support Systems // Proc. of the Ninth International Conference on Enterprise Information Systems. Volume AIDSS. Funchal, Madeira. Portugal. June 12–16. INSTICC, 2007, pp. 277–282.
12. Еремеев А.П., Митрофанов Д.Ю. Методы удовлетворения временных ограничений в интеллектуальных системах поддержки принятия решений реального времени // Программные продукты и системы. 2010. № 1. С. 18–23.
13. Бериша А.М., Вагин В.Н., Куликов А.В., Фомина М.В. Методы обнаружения знаний в «зашумленных» базах данных // Изв. РАН. Теория и системы управления. 2005. № 6. С. 143–158.
14. Vagin V.N., Fomina M.V., Kulikov A.V. The Problem of Object Recognition in the Presence of Noise in Original Data // 10th Scandinavian Conference on Artificial Intelligence SCAI 2008. A. Holst, P. Kruger, and P. Funk (eds.), IOS Press, Amsterdam. 2008, pp. 60–67.
15. Дзегеленок И.И. Открытые интеллектуальные системы // В кн.: Техническое творчество: теория, методология, практика: Энциклопед. слов.-справ.; под ред. А.И. Половинкина, В.В. Попова. М.: НПО «Информ-система», 1995. 408 с.
16. Дзегеленок И.И. Сетевые образовательные технологии актуализации знаний // Информационные технологии в проектировании и производстве: науч.-технич. журн. М.: Изд-во ФГУП ВИМИ. 2003. № 3. С. 10–15.
17. Дзегеленок И.И. Методология поискового проектирования вычислительных систем // Информационная математика. 2004. № 1 (4). С. 110–119.
18. Анисимов Д.Н., Вершинин Д.В., Зуева И.В., Колосов О.С., Хрипков А.В., Цапенко М.В. Использование подстраиваемой динамической модели сетчатки глаза в компонентном анализе для диагностики патологий методами искусственного интеллекта // Вест. МЭИ. 2008. № 5. С. 70–74.
19. Фролов А.Б., Яко Э. Алгоритмы распознавания частично упорядоченных объектов и их применение // Изв. РАН. Техническая кибернетика. 1990. № 5. С. 95–104.
20. Фролов А.Б. Принцип конечной топологии распознавания топологических форм // Изв. РАН. Теория и системы управления. 2010. № 1. С. 68–76.
21. Фролов А.Б., Фролов Д.А., Яко Э. Программируемые функциональные схемы для распознавания упорядоченных объектов // Изв. РАН. Теория и системы управления. 1997. № 5. С. 163–172.
22. Ополченов А.В., Фролов А.Б. Синтез и верификация экспертных систем принятия решений // Изв. РАН. Теория и системы управления. 2002. № 5. С. 101–110.
23. Frolov A.B., Jako E., Mezey P.G. Logical models of molecular shapes and their families // Kluwer Journal of Mathematical Chemistry 30, № 4, Nov. 2001, pp. 389–403.

УДК 681.3.06

ФОРМЫ, ЯЗЫКИ ПРЕДСТАВЛЕНИЯ, КРИТЕРИИ И ПАРАМЕТРЫ СЛОЖНОСТИ ПАРАЛЛЕЛИЗМА

В.П. Кутепов, д.т.н.; В.Н. Фальк, д.т.н.
(Московский энергетический институт (ТУ), vkutepov@appmat.ru)

Рассматриваются различные аспекты создания языков и сред параллельного программирования и реализации параллелизма в компьютерных системах.

Ключевые слова: *параллелизм, параллельные системы, параллельное программирование.*

Компьютер сегодня – не только тонкий инструмент для решения сложнейших научно-технических проблем, но и, пожалуй, главенствующее звено в процессе автоматизации всех сфер челове-

ческой деятельности. Ал-Хорезми, выдающемуся мыслителю и математику Древнего Востока, пытавшемуся найти описание нечто общего и многократно повторяющегося в рациональной деятельности людей, было бы приятно узнать, насколько плодотворной оказалась его идея.

Другому математику и выдающемуся инженеру фон Нейману повезло значительно больше в воплощении алгоритмической парадигмы в реальность: его решение о представлении алгоритма в виде командной программы, хранящейся вместе с данными в памяти компьютера и выполняемой в виде упорядоченной последовательности операций над адресуемыми данными, оказалось настолько простым и плодотворным, что до сих пор продолжает жить в устройстве почти всех компьютеров.

Сегодня нередко можно услышать критику в адрес строго последовательной концепции программы фон Неймана [1] как весьма ограниченной на общем фоне обычно параллельных и асинхронных процессов, с которыми мы сталкиваемся при программировании реальных задач. Но при этом игнорируется тот факт, что модель последовательной программы фон Неймана существенно упрощает архитектуру компьютера, в частности процессорную часть, оставаясь в то же время универсальной в алгоритмическом смысле. Последнее означает, что любые параллельно протекающие конструктивные процессы могут быть представлены в виде последовательной программы с сохранением их функционального значения как однозначных преобразователей входных данных в результаты.

Однако хорошо известно, что последовательные языки оказались совершенно неприспособленными, когда их пытались применить для описания моделирования работы сложных систем, где асинхронность и одновременность выполняемых процессов – норма, а не исключение. Их последовательная операционная семантика весьма ограничительная при программировании вычислений многих семантических объектов, таких, как, например, параллельные функции [2], или при вычислениях, в которых параллелизм является необходимым условием достижения требуемого качества алгоритма. Так, если ставить задачу минимизации времени вычисления значений функции на заданном множестве входных данных, используя при этом множество различных алгоритмов, естественным решением является их одновременное применение к каждому из данных и рассмотрение в качестве результата того из них, которое получено за наименьшее время.

Сегодня компьютерная индустрия может производить компьютерные, или *вычислительные*, *системы* (ВС) с сотнями тысяч самостоятельных компонентов, способных коллективно выполнять сложную работу, будь то вычислительные или управляющие процессы. Поэтому требуются совсем иные модельные, языковые и управляющие средства для их эффективного программирования и последующего параллельного выполнения.

Традиционный подход к решению этой непростой проблемы на основе расширения языков последовательного программирования с целью описания параллелизма и создание распараллеливающих компиляторов, как показывает практика, не дает нужных результатов.

Во-первых, возникает вопрос, зачем естественный параллелизм задач сначала надо (часто с большими усилиями) превращать в последовательные формы описания, а затем выявлять его при компиляции, что не всегда можно сделать в принципе. Во-вторых, какие должны быть языковые средства, чтобы сразу строить программу как параллельную.

Конечно, предыстория развития компьютеров, огромный багаж алгоритмов и программного обеспечения, созданных с использованием языков последовательного программирования, архитектура компьютера как последовательной машины еще долгое время будут обусловливать осторожность при всякой попытке кардинального изменения парадигмы, стиля и языков программирования.

Считается, что история параллелизма началась с работы [3], хотя уже фон Нейман хорошо понимал все ограничения, которые создает концепция последовательного программирования с точки зрения развития архитектуры компьютера и повышения его быстродействия. Опережающие устройства обработки команд в компьютерах 60-х годов *stretch* и БЭСМ-6, динамический анализ и одновременное выполнение независимых в программе команд в CDC-6600, арифметический конвейер, введение векторных команд в компьютерах *Crey* – наиболее значимые архитектурные нововведения, существенно увеличивающие быстродействие современных компьютеров. В среднем компьютер выполняет три-четыре операции одновременно благодаря усовершенствованию его процессора при выполнении последовательных программ.

Векторные команды многопроцессорной ВС ИЛЛИАК-4, возможность задания в последовательной программе ее независимо выполняемых ветвей (в принятой терминологии – нитей), реализованная в системах БЭРРОУЗ и ЭЛЬБРУС, параллельный ФОРТРАН – хорошо известные этапы практического перехода от последовательного программирования к параллельному и организации параллельных вычислений.

Все эти решения, хотя просты и имеют ограниченные возможности с позиции представления параллелизма, интересны тем, что дают средства, адекватные для представления параллелизма на задачном уровне, по крайней мере, вычислительных задач линейной алгебры.

Рекурсия как более мощное средство с точки зрения задания параллелизма, задание упреждающего параллелизма в программе и другие особенности реального параллелизма задач и процессов остались за пределами возможностей простых средств представления параллелизма.

Поскольку, как уже говорилось, так или иначе заданный параллелизм реализуется процессными средствами, управляющими одновременно протекающими при выполнении программы процессами, их взаимодействием, синхронизацией, порождением и др., естественным стало решение о перенесении этих средств в языковую среду (снова в расширение последовательной программы) с целью обеспечения в определенном смысле единообразной и в некотором смысле одноуровневой модели обращения с параллелизмом. Средства *PVM*, *MPI*, *Multithreading* [4, 5] стали стандартными решениями, вообще говоря, достаточно низкоуровневого процессного описания параллелизма в последовательных программах. Да и *OPEN MP* мало что изменяет в этой концепции параллельного программирования, давая возможность программисту с помощью комментариев указывать участки последовательной программы, подлежащие параллельному выполнению.

Радикальный подход к решению проблемы параллелизма в работах по созданию комплексных средств параллельного программирования и управления параллельными процессами на компьютерных системах [6, 7] состоит в комплексном решении трех взаимосвязанных задач.

Первая задача – создание высокоуровневых языков параллельного программирования с формальной денотационной, ориентированной на описание параллелизма на задачном уровне, и формальной операционной семантикой, строго регламентирующей процессы параллельного выполнения программы на компьютерных системах.

В основу такого языка в отличие от языков последовательного программирования может быть положен принцип явного отражения только информационной зависимости по данным между компонентами ее декомпозиции. Как следствие, независимые компоненты становятся источником параллелизма, реализуемого при выполнении программы [8, 9]. Другой способ явного задания состоит в использовании функциональной нотации для отражения параллелизма через характеризацию как параллельных или последовательных операций композиции функций, используемых в языке. По этому принципу построен и реализован на компьютерных системах созданный авторами язык функционального параллельного программирования *FPTL* [10].

Вторая задача – создание среды, позволяющей упростить процесс разработки, отладки, прогонок на компьютерных системах и оптимизации параллельных программ [6, 11]. В отличие от сред поддержки разработки последовательных программ экспериментальное исследование на реальной компьютерной системе параллельной программы с целью ее оптимизации принципиально важно для проектирования качественных и эффективных по времени выполнения и использованию ресурсов параллельных программ.

Наконец, *третья задача* – создание эффективных средств управления параллельными процессами, индуцируемыми при выполнении параллельных программ на масштабируемых и многоплатформенных компьютерных системах.

Эта задача состоит из двух подзадач: оптимального планирования процессов и оптимального управления загруженностью с целью оптимизации использования ресурсов компьютерной системы [6, 7, 11]. Успешно решенная задача (вместе с решением двух предыдущих задач) позволит существенно упростить и сделать независимым от конфигурации компьютерных систем проектирование параллельных программ для сложных вычислительных задач, задач распределенного управления, распределенной обработки информации и др.

Предмет обсуждения в данной статье – существенные для создания высокоуровневых языков параллельного программирования особенности, формы представления и характеристики параллелизма, а также критерии, по которым можно судить о сложности параллельных программ.

Формы и характеристики параллелизма

Понятие параллелизма в вычислениях, управлении, коллективной работе и в других процессах связано с понятием процесса и одновременностью протекания актов и действий в процессах и их взаимодействием.

Поскольку практический интерес представляют процессы, реализующие определенные цели, семантика параллелизма часто имеет внепроцессное обоснование причин, по которым те или иные действия в процессах или сами процессы могут выполняться одновременно.

Систематизируем типы и формы параллелизма, возникающего в процессе решения задач, которые необходимо учитывать при создании методов и языков параллельного программирования и их реализации на ВС.

Параллелизм на процессном уровне

Работу компьютера, его операционной системы сегодня невозможно представить без понятий процесса, одновременности, синхронизации и взаимодействия процессов. Любая электронная схема компьютера построена на основе хорошо выверенных законов, как правило, параллельной работы и методов синхронизации ее взаимодействующих элементов.

По сути многие базисные понятия, используемые в широко известных моделях параллельных процессов – сетях Петри [11], моделях Р. Милнера [12], Ч. Хоара [13] и других, наследуются из языка описания электронных схем.

Последовательная и параллельная композиции процессов, их порождение и взаимодействие, синхронность и асинхронность – наиболее важные понятия, посредством которых описываются различные модели и языки описания параллельно протекающих и взаимодействующих друг с другом процессов.

Именно на базе этих моделей созданы и широко применяются процессные языковые стандарты и средства, используемые для параллельного программирования (*PVM, MPI, Multithreading*).

Для вычислительных задач, обработки информации, распределенного управления эти средства могут оказаться низкоуровневыми и в принципе ненужными, если есть развитые средства для управления процессами (в том числе параллельными) в компьютерных системах. Однако для описания работы операционной системы, процессов функционирования различных систем без этих средств не обойтись.

Очевидно, не все в многообразии поведения процессов может быть непосредственно выражено средствами данных процессных моделей и языковых средств, поскольку они предполагают строго определенные операции композиции элементарных процессов: объединение процессов, последовательную и параллельную композицию процессов, организацию циклов или/и рекурсивные определения как механизмы порождения процессов [12, 13]. Как следствие, переходы из состояния в состояние в процессной модели однозначно определяются локально, на основе непосредственно взаимодействующих процессов. Однако уже такая часто применяемая команда, как *kill* (уничтожение процесса), не является локальной, и ее корректная реализация в принципе невозможна, если скорость порождаемых процессов больше скорости процесса их уничтожения (вдогонку).

Кроме того, анализ корректности параллельного процесса (параллельной программы), означающий отсутствие в нем (в ней) блокировок, гонок и т.п., – чрезвычайно сложная задача [4, 14].

Тем не менее, процессные модели и языки необходимы, если речь идет о реализации параллелизма на практике.

Представление параллелизма на задачном уровне

Декомпозиция и информационная независимость компонентов декомпозиции – основа отражения параллелизма при разработке программ решения сложных задач.

Идея построения языка, в программах которого явно указываются информационные связи (и только они) между ее компонентами (операторами, блоками, модулями), реализована в [8, 9]. Параллелизм в программах на этом языке есть следствие информационной независимости их компонентов, он может быть эффективно реализован средствами программы в ВС [7].

Уточним понятие информационной зависимости компонентов $\mathbf{K_i}$ и $\mathbf{K_j}$, которые вводятся при декомпозиции сложной задачи и ее представлении на этом языке. Будем полагать, что каждому компоненту **К** (оператору, функции и т.п.) однозначно сопоставлено множество **In(K)** входных данных, необходимых для того, чтобы **К** можно было начать выполнять, и множество выходных данных **Out(K)**, которые вычисляет **К**. Семантическое значение того, что делает **К**, пока не затрагиваем.

Определение. Пусть в описании процесса решения задачи компонент $\mathbf{K_j}$ непосредственно информационно зависит от компонента $\mathbf{K_i}$ $(\mathbf{K_i \to K_j})$, если $\mathbf{K_j}$ использует хотя бы одно выходное значение $\mathbf{K_i}$ как входное. Транзитивное замыкание $\xrightarrow{*}$ отношения непосредственной информационной зависимости позволяет говорить об информационной зависимости $\mathbf{K_j}$ от $\mathbf{K_i}$, если $\mathbf{K_i} \xrightarrow{*} \mathbf{K_j}$.

Обозначим $[\mathbf{K_i}]$ множество всех входящих в описание задачи компонентов, которые зависят от $\mathbf{K_i}$, и назовем $[\mathbf{K_i}]$ транзитивным классом $\mathbf{K_i}$.

Если $\mathbf{K_i} \in [\mathbf{K_i}]$, то $\mathbf{K_i}$ определен рекурсивно или принадлежит циклическому участку информационно зависимых от $\mathbf{K_i}$ компонентов, при этом $\mathbf{K_i} \xrightarrow{*} \mathbf{K_i}$.

Определение. Компоненты $\mathbf{K_i}$ и $\mathbf{K_j}$ информационно независимы, если $\mathbf{K_i} \notin [\mathbf{K_j}]$ и $\mathbf{K_j} \notin [\mathbf{K_i}]$.

Языки, в программах которых явно задаются связи, отражающие непосредственную информационную зависимость между их компонентами, и только они, как это сделано в *языке граф-схемного потокового параллельного программирования* (ЯГСПП) [8, 9], позволяют одновременно эксплицировать параллелизм в программе как следствие информационной независимости компонентов. Более того, данный принцип построения программ дает возможность просто реализовать одновременное выполнение компонентов по готовности их входных данных, что практически нельзя эффективно сделать для последовательных программ. Кроме того, легко реализовать также потоковый принцип выполнения программы, когда она применяется к потоку в общем случае поступающих в реальном времени данных на ее входе.

Заметим, что условие независимости компонентов $\mathbf{K_i}$ и $\mathbf{K_j}$ последовательной программы определяется более сложно: $\mathbf{In(K_i) \cap Out(K_j) = \Omega \wedge In(K_j) \cap Out(K_i) = \Omega \wedge Out(K_i) \cap Out(K_j) = \Omega}$, где Ω – пустое множество, а последний член конъюнкции отражает особенность обобществления переменных в программе и использование общей памяти при ее выполнении.

При построении языков, в программах которых информационная зависимость между компонентами задается явно, представление условных конструкций требует особого рассмотрения.

Определение. Допустим, что компонент программы K_j условно зависит от K_i, если K_i определяет необходимость использования выходных данных K_j после выполнения K_i.

На рисунке 1 приведен фрагмент граф-схемы программы, у которой компоненты K_1, K_2, K_3 и K_4 зависят от условия К. Условные входы и выходы компонентов изображаются как квадраты, а другие входы и выходы – в виде точек. При этом, если по левой связи передается значение «истина», будут востребованы результаты выполнения компонентов K_1 и K_2; если значение «истина» передается по правой связи, будут использованы результаты выполнения компонентов K_3 и K_4.

Компоненты K_1 и K_3 для приведенного фрагмента граф-схемы могут выполняться одновременно (при наличии на их входах всех входных данных), причем с упреждением, поскольку только после выполнения К станет ясно, какие выходные данные компонента K_1 или K_3 потребуются при продолжении выполнения программы. В отличие от этого случая K_2 или K_4 будет выполняться только после завершения выполнения компонента К и компонентов K_1 и K_3, от которых они непосредственно информационно зависят.

Определение. Условную зависимость компонентов K_1 и K_3 от К, разрешающую упреждающее выполнение K_1 и K_3, назовем слабой, в отличие от этого условную зависимость K_2 и K_4 от К, разрешающую выполнение K_2 или K_4 только после выполнения К, назовем сильной.

Эквивалентными преобразованиями слабую зависимость можно трансформировать в сильную (см. рис. 2) и наоборот, уменьшая или увеличивая параллелизм в программе.

Ясно, что, увеличивая распараллеливание программы за счет увеличения упреждающих вычислений, в модели параллельного выполнения программы должны быть механизмы явного различения компонентов, отнесенных к упреждающим вычислениям, чтобы всегда отдавать приоритет в выполнении неупреждающим компонентам с целью обеспечения корректности (достижение существующего результата).

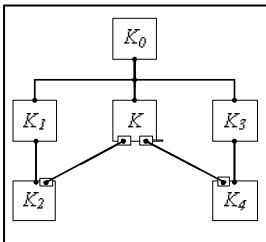

Рис. 1. Пример граф-схемы с условной зависимостью компонентов

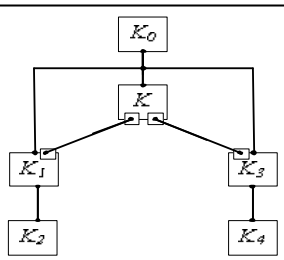

Рис. 2. Условная зависимость компонентов, исключающая возможность упреждающих вычислений

Кроме того, по той же причине и с целью увеличения эффективности необходим механизм прерывания выполняемых с упреждением компонентов после того, как станет известно, что их результаты не потребуются.

Если параллельное выполнение программы организовано таким образом, что выполнение компонентов с упреждением подавляется, то такой параллелизм назовем строгим; в противном случае – упреждающим.

При потоковом параллельном выполнении программы также необходимо учитывать, что временной порядок поступающих на вход граф-схемы данных и вычисленных для них результатов может оказаться нарушенным. Введение в ЯГСПП [6, 9] тегирования данных, передаваемых между компонентами программы, позволяет обеспечивать взаимно однозначное соответствие между входными данными и выходными результатами выполнения программы.

Информационная независимость компонентов программы может выражаться не только явным заданием информационных связей между ними, но также косвенно посредством применения операций композиции компонентов, обладающих свойством параллельности.

Например, в языке *FPTL* используются четыре простые бинарные операции композиции функций, три из которых (*, → и Θ) являются параллельными и только одна операция – суперпозиция (•) – последовательная [10, 11].

К примеру, функция $\mathbf{f(g_1(x), g_2(x))}$ и условный оператор **if p(x) then f_1(x) else f_2(x)** на *FPTL* представляются в виде $\mathbf{(g_1*g_2)\bullet f}$ и $\mathbf{(p\rightarrow f_1)\Theta(p\rightarrow f_2)}$, где операция → имеет семантику условной композиции функций, а операция композиции Θ – семантику объединения графиков ортогональных функций. Семантика операции композиции * – соединение кортежей значений функций, семантика операции • – суперпозиция функций.

В модели параллельного вычисления значений функций на *FPTL* только суперпозиция задает последовательный характер вычисления значений функций, к которым она применяется. Другие три операции композиции параллельны, более того, для операции Θ требуется параллельное или квазипараллельное вычисление значений функций, к которым она применяется.

Фрагмент граф-схемы на рисунке 1 имеет следующее функциональное представление на языке *FPTL*: $\mathbf{K_0\bullet((K\rightarrow K_1)\bullet K_2\Theta(\neg K\rightarrow K_3)\bullet K_4)}$ и может быть приведен путем эквивалентных преобразований к максимально параллельной форме $\mathbf{K_0\bullet((K\rightarrow K_1\bullet K_2)\Theta(\neg K\rightarrow K_3\bullet K_4))}$, где К и ¬К – ортогональные функции, соответствующие двум выходам компонента К на граф-схеме рисунка 1.

Рассмотренные языки параллельного программирования, на которых параллелизм представляется соответствующими конструкциями языка, назовем языками с явным заданием параллелизма.

В языках последовательного программирования параллелизм представлен неявно, и для его реализации нужны, с одной стороны, распараллеливающие компиляторы, выявляющие параллелизм, а с другой – языки параллельного программирования, в которые транслируются последовательные программы.

Коммутативный и некоммутативный параллелизм

Выделение этих форм параллелизма связано с закономерным вопросом: всегда ли компоненты программы, которые могут выполняться одновременно, можно выполнять последовательно в любом порядке. Ответ на этот вопрос имеет принципиальное значение, поскольку ресурсы ВС всегда ограничены и естественны ситуации, когда количество индуцируемых при выполнении программы и способных одновременно выполняться компонентов больше количества процессоров или компьютеров в ВС. В этом случае приходится упорядочивать выполнение этих компонентов, причем избранный порядок не всегда может быть произвольным.

Определение. Параллелизм назовем коммутативным, если допустим произвольный порядок компонентов программы, которые могут выполняться одновременно. В противном случае параллелизм будем называть некоммутативным.

Для условного оператора **if p(x) then f_1(x) else f_2(x)**, хотя и есть возможность одновременного выполнения **p(x)**, f_1**(x)** и f_2**(x)**, однако, если начать вычисление с f_1**(x)**, длящееся неограниченно (функция f_1 не применима к **x**), а значение **p(x)** ложно и f_2**(x)** определено, то невозможно корректно выполнить условный оператор. Вычисление **p(x)** одновременно с f_1**(x)** или f_2**(x)**, как и последовательное выполнение **p(x)**, а затем f_1**(x)** или f_2**(x)**, является корректным.

Сложнее обстоит дело с так называемыми параллельными функциями, корректное вычисление значений которых не может быть просто сведено к последовательной форме.

Рассмотрим пример известной в телефонии параллельной функции голосования **f(g_1(x), g_2(x), g_3(x))** [2], такой, что она определена, если при вычислении значений g_1**(x)**, g_2**(x)** и g_3**(x)** любые два из них определены и равны, причем значением функции **f(g_1(x), g_2(x), g_3(x))** в этом случае является одно из них; в противном случае значение функции не определено. Ясно, что параллельное вычисление значений функций g_1**(x)**, g_2**(x)** и g_3**(x)** при вычислении значения **f(g_1(x), g_2(x), g_3(x))** является естественным, а сведение к простому последовательному вычислению значений в любом порядке не гарантирует получения результата. Достаточно рассмотреть случай, когда вычисление одного из этих значений не определено и длится неограниченно, а два других определены и равны. Только одновременное или квазипараллельное вычисление, например, путем разделения времени вычисления значений g_1**(x)**, g_2**(x)** и g_3**(x)**, может обеспечить корректность вычисления значений функции **f(g_1(x), g_2(x), g_3(x))**. Программирование таких функций на последовательных или параллельных языках требует особого подхода.

В функциональном языке *FPTL* представление и корректное вычисление подобного рода параллельных функций не вызывает проблем.

Приведенная функция голосования в *FPTL* может быть представлена в виде $(g_1*g_2).eq\rightarrow g_1 \Theta \Theta(g_2*g_3).eq\rightarrow g_2 \Theta(g_1*g_3).eq\rightarrow g_3$, где **eq** – бинарная функция проверки равенства аргументов.

Операционная семантика *FPTL* требует, чтобы при вычислении значения функции $f_1 \Theta f_2$, полученной путем Θ-композиции ортогональных функций f_1 и f_2 (заметим, что $f_1 \Theta f_2$ – параллельная функция), вычисление значений f_1**(x)** или f_2**(x)** не откладывалось на неограниченное время, что достигается поочередным выделением им процессорного времени.

Кроме того, параллельные функции относятся к функциям с некоммутативным параллелизмом. В отличие от упреждающего параллелизма, когда значения компонентов программы, которые должны выполняться с упреждением, можно откладывать, для параллельных функций, как уже было сказано, это может приводить к тому, что вообще не будет получен результат выполнения программы.

Потоковый параллелизм множества данных

Природа информационной независимости компонентов программы как необходимое условие ее распараллеливания рассмотрена выше.

Потоковый параллелизм имеет, скорее, организационную основу и восходит к конвейерной обработке, которая предполагает совмещение нескольких этапов последовательной обработки деталей (начало этому положил конвейерный способ организации технологического процесса, предложенный в начале прошлого века Тейлором). Конвейерный принцип выполнения команд в опережающем устройстве компьютера или операций в его арифметическом устройстве – примеры реализации конвейера в компьютерах.

Организуя поток данных на входе последовательной программы, получим классическую конвейерную схему, когда любой компонент программы, завершив выполнение поступивших данных, переходит к обработке следующих данных входного потока.

Линейный конвейер естественным образом можно расширить, введя разветвленную схему параллельной обработки, представляемую, например, в виде граф-схемы, как это сделано в ЯГСПП. Таким образом, легко достигается объединение параллельных процессов, индуцируемых при выполнении информационно независимых компонентов программы и разветвленной конвейерной (потоковой) обработки. Это существенно упрощает разработку параллельных программ для задач реального времени, в частности, распределенных управляющих систем.

Вместе с тем нет необходимости на каждой стадии конвейера обрабатывать только одну порцию входного потока данных. Можно использовать схему одновременного применения компонента программы к множеству всех поступивших на его вход данных, если для этого есть свободные процессоры или компьютеры в ВС.

Эта форма параллелизма реализована в ЯГСПП и в классификации Флинна соответствует способу организации параллельной обработки *SIMD*: один поток команд и множество потоков данных.

Другие схемы параллельной обработки (*SISD, MISD, MIMD*), введенные Флинном, скорее, раскрывают архитектурные особенности ВС, которые эти схемы параллельной обработки реализуют. *SISD* – последовательная обработка, *MISD* – типичная схема обслуживания запросов к БД, а *MIMD* – это то, что присуще работе любой многомашинной и многопроцессорной ВС.

Асинхронный и синхронный параллелизм

Оба эти понятия отражают отсутствие или наличие временных ограничений, накладываемых на следование событий в процессах, реализующих вычисления или управление.

Определение. Процесс, происходящий таким образом, что не накладывается никаких ограничений на длительность протекающих в нем актов (длительность выполняемых компонентов программы), называется асинхронным. Он также предполагает, что выполнение любого акта начинается сразу при выполнении условий, вытекающих из причинно-следственных связей его с другими актами, определяющими его готовность к выполнению. Если это условие нарушается, например, искусственно увеличивается время выполнения операторов, входящих в векторную конструкцию, то такой процесс называется синхронным.

Введение синхронизации часто упрощает описание и реализацию параллельных вычислений, как это очевидно для векторных и матричных задач. Описание и реализация асинхронных процессов существенно усложняют и то, и другое. Для этого достаточно ознакомиться с методами построения асинхронных схем, базирующихся на апериодических автоматах [15], или с моделью асинхронных вычислений значений функций на языке *FPTL*.

Заметим, что понятия асинхронности и параллельности не тождественны. Например, реализация процессов выполнения последовательной программы может быть асинхронной и в то же время не содержать параллелизма.

Параллелизм и проблемная среда

По-видимому, полезным может быть рассмотрение специфики задачных сред и особенностей построения в них параллельных алгоритмов и программ.

Очевидно, матричные задачи, сеточные схемы решения уравнений в частных производных и другие задачи линейной алгебры просты в распараллеливании, и параллельный Фортран, языки *PVM*, НОРМА, *mpC* и другие приспосабливались к этому типу задач. Стандарт *OpenMP* полезен для ручного распараллеливания программ, особенно, если они написаны для этого круга задач.

Совсем иные языковые средства требуются для описания сложных процессов, возникающих, например, в работе ВС, управлении, в системах массового обслуживания, для которых асинхронность и параллелизм являются неотъемлемыми свойствами. ЯГСПП создавался с ориентацией на эффективное параллельное программирование этого круга задач.

Поиск адекватных моделей и языковых средств, предназначенных для эффективного описания и реализации параллелизма в задачах различных проблемных сред, находит отражение в создании проблемно-ориентированных языков параллельного программирования: функциональных *FPTL*, *Haskell* [16], *ML* [17] и др., процессных *MPI, Multithreading*, логических *Parlog* и др.

Критерии и параметры оценки параллелизма

Процесс программирования можно рассматривать как многоэтапный переход от задачи к представлению выбранного метода ее решения на конкретном языке программирования. Показатели, характеризующие качество разработанной программы, степень достижения предъявляемых к программе требований (времени вычисления, объему требуемой памяти и др.), в большей степени определяются методом решения задачи, выбором языка программирования и собственно уровнем программистского искусства, которое заключается в умелом использовании языковых средств для точного представления метода решения задачи и построения эффективной программы.

Для оценки сложности и качества параллельных программ используется целый ряд критериев и параметров.

Коэффициент ускорения

Коэффициент ускорения имеет принципиальное значение и обычно определяется как отношение времени выполнения параллельной программы на одном компьютере **t(1)** ко времени ее выполнения **t(N)** на ВС с N компьютерами или процессорами: **C(N)=t(1)/t(N)**.

Согласно закону Амдала предельное ускорение решения задачи в параллельной форме на ВС с N узлами определяется соотношением: **C(N)= =1/(α+(1-α)/N)**, где **α** – доля операций в программе, которые выполняются последовательно. Предельное ускорение при неограниченном N, очевидно, равно **1/α**. В реальности ускорение меньше предельного из-за затрат времени на управление параллельным выполнением программы на ВС, реализацию обмена данными.

На практике более важно понять, как ведет себя ускорение в зависимости от сложности задачи и количества компьютеров (процессоров) N в ВС.

Пусть $C(x_1, x_2, \ldots, x_k, N)$ есть функция, определяющая ускорение решения задачи в параллельной форме в зависимости от параметров $x_1, x_2, \ldots, x_k$ сложности задачи и количества узлов N в ВС.

Для многих задач часто достаточно лишь определить предельное значение $C(x_1, x_2, \ldots, x_k, N)$, если предполагать, что количество компьютеров или узлов ВС не ограничено (в этом случае не возникает задержек при выполнении параллельной программы из-за отсутствия необходимых ресурсов), и не учитывать системные издержки на организацию выполнения параллельной программы на ВС (обменные взаимодействия, управление и др. [7]). Именно при этих предположениях обычно определяется коэффициент ускорения для многих методов и параллельных программ.

В качестве параметров $x_1, x_2, \ldots, x_k$, характеризующих вычислительную сложность задачи, во многих случаях используются параметры, определяющие размерность задачи, которые одновременно являются и аргументами программы, представляющей метод ее решения.

Рассмотрим пример простой задачи вычисления значений **n!**, используя для этого параллельный метод разбиения отрезка **[1÷n]** пополам, по программе: $\mathbf{Fact(i, j)=1\ if\ i=j\ else\ Fact(i,\lfloor(i+j)/2\rfloor)\times Fact(\lfloor(i+j)/2\rfloor+1, j))}$, где $\lfloor\alpha\rfloor$ – ближайшее целое к **α**. Очевидно, что **F(1, n)=n!**.

Если попытаться реализовать процесс вычисления значения **n!** по этой программе, используя все возникающие возможности распараллеливания, нетрудно показать, что этот процесс следует схеме, изображенной на рисунке 3.

Параметр **n** (аргумент приведенной выше функции) характеризует вычислительную сложность задачи, а коэффициент **C(n)** (при неограниченном числе узлов ВС) ведет себя, как $\mathbf{O(n/\log_2 n)}$, и при неограниченном увеличении **n** стремится к бесконечности.

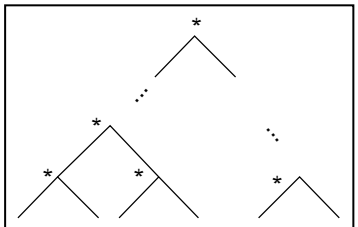

Рис. 3. Схема параллельного вычисления n!

Таким образом, с усложнением задачи (увеличением **n**) можно неограниченно сокращать время выполнения этой программы при условии неограниченных ресурсов.

Можно привести множество примеров других задач (перемножение матриц, решение систем линейных уравнений и др.), для которых с увеличением их сложности коэффициент ускорения возрастает неограниченно.

Параллельные программы такого типа назовем программами с неограниченным параллелизмом.

Глубина и степень распараллеливания

Введем еще один параметр, определяющий глубину распараллеливания метода (или представляющего его алгоритма) и интуитивно характеризующий в среднем вычислительную сложность компонентов параллельной программы, которые рассматриваются как ее самостоятельные части и могут выполняться одновременно. В литературе этот параметр часто называется зернистостью параллелизма.

Определение. Глубиной распараллеливания **d** назовем усредненную вычислительную сложность компонентов, которые рассматриваются в параллельной программе как самостоятельные процессы, идентифицируемые и планируемые при ее выполнении на ВС.

Обратную к **d** величину определим как степень распараллеливания, характеризующую усредненное количество компонентов параллельной программы, которые могут выполняться одновременно.

Рассмотрим простой пример перемножения квадратных матриц размера **n** и определим предельный коэффициент ускорения процесса параллельного решения этой задачи для различных **d** и неограниченного количества компьютеров в ВС:

1) $\mathbf{d_1}$ – сложность вычисления одной строки результирующей матрицы, а степень распараллеливания – одновременное вычисление всех строк матрицы: $\mathbf{C(n, d_1)} = \dfrac{\mathbf{n^2\left(n\times t_{ум}+(n-1)\times t_{сл}\right)}}{\mathbf{n\left(n\times t_{ум}+(n-1)\times t_{сл}\right)}} = \mathbf{n}$,

где $\mathbf{t_{ум}}$ и $\mathbf{t_{сл}}$ – время выполнения операций умножения и сложения соответственно;

2) $\mathbf{d_2}$ – сложность вычисления одного элемента результирующей матрицы, а степень распараллеливания – одновременное вычисление всех элементов результирующей матрицы:

$$C(n, d_2) = \frac{n^2 \left(n \times t_{ум} + (n-1) \times t_{сл}\right)}{n \times t_{ум} + (n-1) \times t_{сл}} = n^2;$$

3) d_3 – сложность операции умножения, а степень распараллеливания – одновременное выполнение всех операций умножения при одновременном вычислении всех элементов результирующей матрицы:

$$C(n, d_3) = \frac{n^2 \left(n \times t_{ум} + (n-1) \times t_{сл}\right)}{t_{ум} + (n-1) \times t_{сл}} = O\left(r_1 \times n^3\right),$$

где r_1 – некоторая константа, $r_1 \le 1$; $O(x)$ – близкое к x значение;

4) d_4 – усредненная сложность выполнения операции умножения и параллельного вычисления суммы $\sum_{k=1}^{n} M'[i,k] \times M''[k,j]$ при вычислении элементов результирующей матрицы; степень распараллеливания определяется исходя из условия одновременного выполнения всех операций умножения при последующем параллельном вычислении приведенной выше суммы делением отрезка $[1 \div n]$ пополам (см. пример параллельного вычисления факториала выше):

$$C(n, d_4) = \frac{n^2 \left(n \times t_{ум} + (n-1) \times t_{сл}\right)}{t_{ум} + \log_n n \times t_{сл}} = O\left(r_2 \times n^3\right),$$

где $r_2 \le r_1$.

Нетрудно заметить, что во всех четырех случаях коэффициент ускорения неограниченно растет с увеличением n. Однако производная (ускорение) этого роста различна для разных d, например, для d_1, d_2, d_3 коэффициент ускорения растет как $O(n)$, $O(n^2)$ и $O(r_1 \times n^3)$ соответственно.

Обратим внимание, что степень распараллеливания ведет себя как неубывающая функция в зависимости от величины $1/d$ и всегда существует ее предельное значение, определенное предельной и всегда ограниченной глубиной распараллеливания.

Варьирование степени распараллеливания задачи чрезвычайно важно при оптимизации процесса выполнения ее на ВС с позиции минимизации времени выполнения и использования ресурсов. Это может происходить как на стадии разработки параллельной программы и ее статическом планировании на ВС, так и на стадии выполнения, когда динамически варьируется степень распараллеливания с целью увеличения фронта готовых к выполнению процессов, что ведет к увеличению загруженности ВС.

Интенсивность обменных взаимодействий

Эффективность параллельной работы ВС также существенно зависит от пропускной способности коммуникаций и интенсивности обменных взаимодействий между ее компонентами в процессе выполнения параллельной программы. Поэтому важно понять, каким образом при изменении степени распараллеливания будет изменяться интенсивность обменных взаимодействий при выполнении программы на ВС.

Определение. Определим интенсивность обменных взаимодействий для параллельной программы при глубине распараллеливания d как функцию $\lambda(\bar{x}, d, N) = n / t(N)$, где n – количество обменных взаимодействий при выполнении параллельной программы заданной сложности $\bar{x} = x_1, x_2, .., x_k$ на ВС с N узлами.

Если количество узлов ВС N таково, что задержек при выполнении параллельной программы не существует из-за недостатка компьютеров или процессоров в ВС, будем обозначать $\lambda(\bar{x}, d)$ как предельное значение интенсивности обменов.

Легко проверить следующее: для рассмотренных примеров параллельного вычисления $n!$ и перемножения матриц, если полагать, что они осуществляются на ВС с разделенной памятью, $\lambda(\bar{x}, d)$ ведет себя в зависимости от $\bar{x}$ и d практически так же, как и степень распараллеливания.

При увеличении степени распараллеливания до определенного предела путем изменения d время выполнения параллельной программы сначала уменьшается. Однако при этом увеличивается интенсивность обменов между узлами ВС и при $d < d_{опт}$ начинается естественное замедление вычислений из-за увеличения издержек на их реализацию.

Может показаться, что параллельные вычисления на ВС с общей памятью устраняют проблему снижения эффективности их работы из-за обменных взаимодействий между компьютерами. Однако достаточно рассмотреть пример вычисления значений функции $n!$ в абсолютно параллельной форме согласно рисунку 1, чтобы понять: при реализации этой стратегии к общей памяти одновременно могут обращаться порядка $n/2$ команд. С увеличением n ограниченная пропускная способность системы «процессоры–память» станет узким местом, что существенно уменьшит эффект от распараллеливания.

Более того, для параллельной программы, как правило, требуется гораздо больший объем оперативной памяти (в рассмотренном примере порядка $n/2$ ячеек памяти), а возникающие обмены с дисковой памятью могут свести к нулю эффект от распараллеливания.

Использование ресурсов

На практике при организации параллельных вычислений важно знать не только ускорение, получаемое при выполнении параллельной программы на реальной ВС, но и то, как при этом используются ее ресурсы.

Для определения эффективности применения ресурсов обычно используют отношение $C(x_1, x_2, \ldots, x_m, N)/N$, причем часто считают, что это отношение не может быть больше единицы, полагая, что ускорение не может быть больше N – количества узлов, процессоров и компьютеров ВС.

В действительности это не так, поскольку для сложных задач, требующих большого объема памяти, ускорение может быть больше N. Это связано с тем, что для сложных задач время выполнения программы на одном компьютере (предполагая, что его память не больше объема памяти всех компьютеров ВС) может существенно увеличиться из-за большой интенсивности обменов между оперативной памятью и дисковой.

Более точное представление о реальном использовании ресурсов, в частности узлов ВС, дает усредненное значение их загруженности на интервале **T** выполнения параллельной программы: $\sum_{i=1}^{N} \frac{1}{T} \int_0^T Li(t)dt$, где **Li(t)**, **i=1, 2, …,** N – загруженность **i**-го узла. Обычно о загруженности ВС судят по загруженности только ее процессоров, что часто оправдано для вычислительных задач.

Однако задача может быть такой, что в ее программе основную часть времени занимает работа с периферией (ввод/вывод) или обработка сложных массивов данных. В этих случаях акцент смещается в сторону организации эффективного управления памятью, портами, периферией и др. В [7] при исследовании проблемы управления параллельной работой ВС показано, что ее эффективность существенно зависит от интенсивности индуцируемых при выполнении программ команд ввода/вывода, обмена с дисковой памятью, в том числе из-за возникающего обмена страницами между оперативной и дисковой памятью по причине недостатка первой, а также из-за обмена данными между узлами.

Если степень загруженности любого из указанных трактов, то есть действующего в нем оборудования, определяется как отношение λ/μ, где λ – интенсивность входного потока, μ – интенсивность его обслуживания, и это отношение оказывается больше единицы, данный тракт становится узким местом, резко снижающим общую производительность параллельной работы ВС.

Интересно проследить общий характер изменения показателя эффективности использования ресурсов ВС для задачи заданной сложности с увеличением N – количества узлов ВС.

Очевидно, если не изменяется глубина распараллеливания и сохраняется общая схема организации выполнения параллельной программы на ВС, этот показатель сначала должен увеличиваться (точнее, не уменьшаться) с увеличением N. Однако всегда будет существовать некоторое оптимальное значение N, больше которого уже нельзя уменьшить время выполнения программы (нужна задача большей сложности или требуется уменьшение глубины распараллеливания).

Можно сделать вывод, что при построении эффективных параллельных программ необходимо не только уделять серьезное внимание разработке параллельных методов решения задач, но и уметь их анализировать и приспосабливать к масштабу ВС и ее техническим возможностям для достижения максимального эффекта.

Подобно всяким изобретениям языки программирования, конечно, имеют прикладное значение и их создание и развитие обусловлены как реальной практикой применения, так и бурным развитием вычислительной техники. Состоявшийся долгожданный переход компьютерной индустрии к широкому производству компьютерных систем (кластеров, сетей и т.п.), насчитывающих тысячи и даже сотни тысяч компонентов, заставляет разработчиков срочно решать задачу создания языковых и управляющих средств для эффективного применения такого рода систем. В статье авторы попытались обозначить реперные точки, вокруг которых происходит объективное столкновение различающихся и часто противоположных подходов к созданию языков и сред параллельного программирования.

Очевидно, что созданное фирмами-разработчиками программное обеспечение пока далеко от того, чтобы сделать эффективными процессы разработки параллельных программ и их выполнение на различных компьютерных системах.

Уже разработано огромное количество языков последовательного программирования и операционных средств, и понятно, что объектно-ориентированное программирование – лишь очередной этап развития. Скорее всего, это касается создания и повышения уровня языков и сред параллельного программирования, а также создания теоретической базы и практической реализации методов и средств управления процессами в больших компьютерных системах.

Литература

1. Backus J. Can programming be liberated from the von Neuman style? Communication of ACM, 1978, № 1, pp. 613–641.

2. Трахтенброт Я.Б. Обогащение алгоритмических языков параллельными функциями: Автореф… к.ф.-м.н. Н.: ИМ СО АН СССР, 1978.

3. Gill S. Parallel programming. The computer journal, 1958, № 1, pp. 2–10.

4. Грегори Эндрюс Р. Основы многопоточного параллельного и распределенного программирования. М.: Изд. дом «Вильямс», 2003. С. 1–506.

5. Хьюз К. и Хьюз Т. Параллельное и распределенное программирование. М.–СПб–К., 2004. С. 1–667.

6. Кутепов В.П. Об интеллектуальных компьютерах и больших компьютерных системах нового поколения // Изв. РАН. ТиСУ. 1996. № 5.

7. Кутепов В.П. Интеллектуальное управление процессами и загруженностью в вычислительных системах // Изв. РАН. ТиСУ. 2007. № 5. С. 58–73.

8. Котляров Д.В., Кутепов В.П., Осипов М.А. Граф-схем-

ное потоковое программирование и его реализация на компьютерных системах // Изв. РАН. ТиСУ. 2005. № 1. С. 75–96.

9. Kutepov V.P., Malanin V.M., Pankov N.A. An approach to the development of programming software for distributed computing and information processing systems. ICSOFT-08, International conference on software and data technologies, Porto, Portugal, 2008, pp. 83–90.

10. Бажанов С.Е., Кутепов В.П., Шестаков Д.А. Язык функционального параллельного программирования и его реализация на кластерных системах // Изв. РАН. Программирование. 2005. № 5.

11. Бажанов С.Е., Воронцов М.Н., Кутепов В.П. Структурный анализ и планирование процессов параллельного выполнения функциональных программ // Изв. РАН. ТиСУ. 2005. № 6. С. 111–126.

12. Milner R. A calculus for communicating systems. Lecture notes in computing science. Springer – Verlag, New York, 1980. Vol. 92.

13. Хоар Ч. Взаимодействующие последовательные процессы. М.: Из-во «Мир», 1989. С. 240.

14. Журнал для разработчиков MSDN Magazine. М.: 2008, 11(83). URL: www.microsoft.com/rus/msdn/magazine (дата обращения: 16.04.2010).

15. Апериодические автоматы, под ред. В.И. Варшавского. М.: Наука, 1976. С. 424.

16. Jones S.T. Peyton. The implementation of functional programming languages. Prentice Hall, 1987.

17. Milner R. The standard ML core language. Polymorphism // The ML/LCF/ Hope Newsletter, October, 1985. Vol. 2. № 2.

УДК 004.8

МЕТОД ПОСТРОЕНИЯ НЕЧЕТКОЙ ПОЛУМАРКОВСКОЙ МОДЕЛИ ФУНКЦИОНИРОВАНИЯ СЛОЖНОЙ СИСТЕМЫ

(Работа поддержана РФФИ, грант 10-01-97506-р_центр_а)

Ю.Г. Бояринов, к.т.н.; В.В. Борисов, д.т.н.; В.И. Мищенко, д.т.н.; М.И. Дли, д.т.н.
(Смоленский филиал Московского энергетического института (технического университета), BYG@yandex.ru)

Анализируется функционирование систем на основе нечетких полумарковских моделей. Рассматриваются различные способы введения нечеткости в полумарковские модели в зависимости от характера используемой информации и особенностей решаемых задач анализа функционирования систем.

Ключевые слова: *нечеткая полумарковская модель, нечеткий вывод, нечеткая функция, нечеткое отображение.*

Одним из эффективных инструментов анализа функционирования систем, базирующихся на вероятностном подходе, является использование полумарковских моделей для оценки вероятностей нахождения системы в различных состояниях.

Сложность полумарковской модели функционирования системы определяется как множеством учитываемых факторов, так и непростой организацией самой системы (наличием разнородных подсистем, элементов и взаимосвязей между ними). Существенным является также необходимость учета фактора неопределенности анализируемых переменных и случайности событий.

Традиционно используемый для учета стохастической неопределенности вероятностный подход в полумарковских моделях не всегда применим из-за недостатка статистической информации о состоянии сложной системы. Кроме того, при традиционном подходе невозможно учесть:

- неопределенность переменных, обусловленную экспертным характером значительной части информации или эвристическим описанием процессов;
- разнокачественность данных, а также их оценку с помощью различных шкал;
- нечеткость выделения и описания переменных или отдельных состояний, а также входных и выходных воздействий.

Одним из основных способов решения подобных проблем при анализе функционирования систем является использование нечетких полумарковских моделей [1–3].

Следует отметить определенные ограничения существующих нечетких полумарковских моделей:

- не всегда четко обосновано соответствие вложенных цепей полумарковской модели классу цепей Маркова;
- имеющиеся модели в общем случае используются только для прогнозирования характеристик пребывания моделируемой системы в заданном состоянии, что затрудняет решение задач управления;
- предлагаемые подходы к построению нечетких полумарковских моделей не учитывают наличие избыточности ресурса как необходимое условие функционирования сложной системы.

Использование полумарковских моделей для анализа функционирования систем

Сложные системы, как правило, характеризуются некоторым уровнем избыточности, позволяющим, с одной стороны, противостоять отказам (или другим повышающим их энтропию явлениям), с другой стороны, накапливать соответст-