НЕКОТОРЫЕ АСПЕКТЫ ПРОГРАММИРОВАНИЯ И УПРАВЛЕНИЯ ВЫЧИСЛИТЕЛЬНЫМИ ПРОЦЕССАМИ В МАШИНАХ С ДИНАМИЧЕСКОЙ АРХИТЕКТУРОЙ

И. В. Царев

Санкт-Петербургский институт информатики и автоматизации РАН
199178, Санкт-Петербург, 14-я линия В.О., д.39
civ@mail.iias.spb.su

УДК 681.3.06

И. В. Царев. **Некоторые аспекты программирования и управления вычислительными процессами в машинах с динамической архитектурой** // Труды СПИИРАН. Вып. 1, т. 1, — СПб.: СПИИРАН, 2002.

Аннотация. *Рассматриваются методы и средства организации программ в мультипроцессорной среде с динамической архитектурой. Предлагается объектно-ориентированный метод программирования автотрансформирующихся сетевых программ, отражающих в большей степени структуру решаемой задачи, чем свойства вычислительной среды. Обсуждаются средства поддержки данного подхода на аппаратном уровне и уровне операционной системы и метод графического проектирования параллельно выполняемых программ в сетевом представлении* — Библ. 7 назв.

UDC 681.3.06

I. V. Tsaryov. **Some viewpoints on programming and control of computational processes in computers with dynamic architecture** // SPIIRAS Proceedings. Issue 1, v. 1, — SPb: SPIIRAS, 2002.

Abstract. *Methods and means for organizing programs in multiprocessor environment with dynamic architecture are considered. Object-oriented method for programming autotransforming network-represented programs that reflect mainly the structure of the problem solved but properties of computational environment is suggested. Means for supporting this approach on hardware level and on the level of the operating system and the method of graphical parallel program design in the network representation are discussed.* — Bibl. 7 items.

1. Введение

Возможности первых ЭВМ были весьма ограничены, поэтому многие архитектурные решения разрабатывались с целью минимизировать и оптимизировать набор аппаратных компонентов и систему команд машины, сводя набор операций к теоретическому минимуму, при этом одним из главных оснований для этих решений был выбор некоторого ограниченного набора операций, основанных на Булевой алгебре и классической машинной арифметике, т.е. арифметическое устройство (АУ) и устройство управления (УУ) выполняли некоторый набор общепринятых операций, хотя конкретные наборы операций, их структура и кодировка в различных машинах существенно различались. Именно тогда возникло понятие «программирование для ЭВМ такого-то типа». Эта фраза стала настолько привычной, что не сразу приходит в голову ее парадоксальность. Не «ЭВМ для решения задач» и не «ЭВМ для программирования алгоритмов», и даже не «программирование для решения задач на ЭВМ», как будто главным элементом цепочки «задача – метод решения – алгоритм – программа – ЭВМ» является именно ЭВМ. Причина этого, конечно, ясна. Ограниченные возможности аппаратуры требовали минимизации базовых функций ЭВМ, приведения их к некоторому минимальному набору операций. Теоретические основания ЭВМ, такие как машины фон Неймана, Булева алгебра и т.п. окончатель-

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

но определяли и архитектуру ЭВМ и набор операций. Это было тем более оправдано, что ЭВМ использовались преимущественно для вычислений, что и нашло отражение не только в их архитектуре, но и в названии (как в русском, так и в других языках). Однако, попытки приблизить машину если не к структуре самой задачи, то хотя бы к структуре алгоритма и принятых в программах структур данных и операторных структур, неоднократно предпринимались. Сначала появились языки программирования, которые отделили алгоритм и программу от архитектуры и системы команд конкретной ЭВМ, затем – операционные системы, которые взяли на себя значительную долю проблем, связанных с конкретной аппаратурой и управлением вычислительными процессами. Позднее появились и машины (или процессоры), ориентированные либо на конкретные языки программирования, либо на общепринятые программные структуры. К первым можно отнести разного рода ЛИСП-процессоры, ФОРТ-процессоры и т.п., а ко вторым — мощные проекты типа отечественных машин «Эльбрус» и аналогичных американских машин фирмы “Burroughs”. К сожалению, подобные решения не получили большого распространения, и во многом были не слишком удачными. К тому же следует отметить, что они опять-таки были ориентированы не на структуры, адекватные решаемым задачам, а на уже сложившиеся к тому времени структуры языков программирования, которые в значительной мере определялись традиционной архитектурой компьютеров (последовательное выполнение команд, единое линейно организованное пространство памяти и др.). Противоречие между структурой задач, алгоритмов и архитектурой компьютеров, известное как «семантический разрыв», было практически безвозвратно закреплено в последней. Дальнейшее развитие архитектуры и технологии производства вычислительных устройств составляет целую историю борьбы с неэффективностью вычислительного процесса, одной из самых значительных причин которой как раз и является этот «семантический разрыв». Однако, эта борьба велась «на поле противника», т.е. не посредством приближения архитектуры компьютеров к структуре задачи, а постоянными попытками скомпенсировать сложившиеся хронические проблемы. Все большая часть аппаратуры работала не на выполнение тех операций, которые составляют сущность решения задачи, а на то, чтобы ускорить выполнение непроизводительной работы. Кэширование памяти, предсказание переходов, конвейеризация, этот список грозит стать слишком длинным, чтобы его продолжать. Хочется лишь упомянуть некоторые из последних «архитектурных перлов» – динамическое переименование регистров и попытка выполнять операции не в том порядке, как они написаны, да еще несколько операций одновременно. Появление технологии БИС первоначально перечеркнуло даже те архитектурные достижения, которые уже существовали к тому времени, в силу того, что архитектурные возможности первых микропроцессоров были еще более ограничены, чем у первых «мэйнфреймов». В дальнейшем это было закреплено благодаря требованиям программной совместимости. В результате в современном процессоре из нескольких десятков миллионов логических элементов и вентилей более половины борются с низкой эффективностью работы остальных, но результаты этой борьбы не впечатляют.

Идея параллельного выполнения разных частей программы на различных блоках аппаратуры (не обязательно на полноценных процессорах), возникшая сравнительно давно, тем не менее, оказалась в плену у сложившихся архитектурных решений, поэтому недостатки, присущие «семантическому разрыву», были тиражированы в мультипроцессорных компьютерах, а к уже имеющимся

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

проблемам добавились новые – как распараллеливать программу, как обмениваться потоками данных между процессорами, как синхронизировать эти процессы и т.д. Результат также неутешителен. Мультипроцессоры с объявленной производительностью в миллиарды и триллионы операций в секунду реально показывают подобную производительность только на узком классе специально отобранных регулярных задач, а на многих других — падает в десятки и сотни раз, да и такие «достижения» требуют поистине титанических усилий от разработчиков прикладных программ, трансляторов и операционных систем. Средства параллельного программирования весьма убоги и сводятся, в основном, к расширению традиционных последовательных языков программирования процедурами порождения и уничтожения параллельных процессов, их синхронизации и обмена данными (сообщениями) между ними, причем управлять этими сложными действиями должен программист, что требует огромной квалификации и опыта.

Но существует другой, нетрадиционный путь, основные особенности которого состоят в динамической перестройке архитектуры компьютера в зависимости от структуры решаемой задачи, разделении процессов вычисления (т.е. преобразования информации в соответствии с вычислительным методом и алгоритмом), управления (принятия решений о ходе вычислительного процесса) и коммутации (порождение процессов и их распределение по вычислительным ресурсам), полной децентрализации управления, асинхронном выполнении параллельных вычислительных процессов, отказе от ряда других традиционных решений. Этот путь нашел свое воплощение в идее мультипроцессоров с динамической архитектурой (МДА), которая развивается в течение более двадцати лет в Санкт-Петербургском институте информатики и автоматизации РАН (СПИИРАН) под руководством проф. В. А. Торгашева. Особенности организации параллельных вычислительных процессов и их программирования в МДА рассматриваются в настоящей статье.

2. Основы МДА

Большинство задач, решаемых на ЭВМ, в той или иной степени обладают «естественным параллелизмом». Это означает, что практически любая достаточно сложная задача может быть разбита на небольшие функционально законченные фрагменты, многие из которых не зависят друг от друга (под «задачей» в данном случае понимается не абстрактная математическая задача, а реальная практическая задача, которая, как правило, включает в себя множество различных вычислений, а также различные преобразования информации, отображение результатов, машинная графика и многое другое). Общепринятый алгоритмический подход к решению задач предполагает изначально последовательный характер выполнения программы, который затем, при наличии множества вычислительных ресурсов, приходится искусственно принудительно «распараллеливать». Т.е. сначала выстраивается последовательная цепочка действий, совсем необязательно естественная для данной задачи, а потом ее «перекраивают» для параллельного выполнения. В то же время все мы постоянно сталкиваемся с тем, что в природе все процессы происходят независимо и параллельно, взаимодействуя только там, где это неизбежно. Нам немногое известно о функционировании мозга, однако очевидно, что мозг легко и быстро справляется со сложнейшими задачами, такими как распознавание образов, поиск информации в памяти, логический вывод и принятие решений, управле-

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

ние движениями тела, и многими другими, несмотря на то, что физическое быстродействие элементов (нейронов) на много порядков ниже, чем быстродействие современных электронных чипов. Это происходит из-за того, что миллиарды нейронов работают одновременно и в значительной мере независимо друг от друга.

Естественным представлением любых процессов, происходящих в природе, технических системах, биологических организмах или в обществе является представление в виде сети, состоящей из множества разнородных объектов, между которыми существуют связи различных типов, причем в процессе функционирования меняется и количество объектов разных типов, и связи между ними. Аналогичным образом можно представить и практически любую решаемую задачу.

Более 20 лет назад В. А. Торгашевым была выдвинута идея рекурсивных вычислительных машин (РВМ) [1], в середине 70-х годов реализованная в виде макетного образца. Анализ недостатков РВМ и дальнейшее развитие идеи привело в начале 80-х годов к созданию теории динамических автоматных сетей (ДАС) [2, 3], которая была положена в основу МДА. Современный взгляд на МДА изложен в [4]. Кратко суть этой идеи заключается в следующем. Имеется некоторое (теоретически неограниченное) множество свободных динамических автоматов (ДА), обладающих различными свойствами и функциями, а также некоторое множество коммутационных автоматов, связанных между собой в коммутационную автоматную сеть (КАС). Управляющее воздействие на КАС осуществляет программа, которая представляет собой сеть из связанных между собой ДА (изначально программа может состоять из одного ДА). КАС является лишь средой для функционирования ДА, которая «помогает» им выполнять их «коммутационные» функции. К последним относятся, например, выбор подходящего по свойствам свободного ДА и включение его в программную сеть (это называется «порождением» автомата), отсоединение автомата от ДАС и его возвращение в множество свободных ДА («уничтожение» автомата), изменение связей между автоматами. Каждый автомат обладает рядом свойств, включая состояние, которые и определяют его поведение в сети, например порождение и уничтожение, а также выполнение некоторой «вычислительной» функции. Последняя, помимо свойств и состояния самого ДА, определяется и состоянием его ближайших соседей, т.е. ДА, с которыми он непосредственно связан. ДА никак не зависит от других ДА, с которыми он не связан. Автомат, выполнивший свою основную функцию, как правило, уничтожается (хотя возможно и иное). Таким образом, получается сеть из автоматов, которая в процессе выполнения ими своих функций непрерывно изменяется, как по числу и свойствам ДА, так и по конфигурации связей между ними (это свойство называется автотрансформацией ДАС). В какой-то момент ДАС может приобрести стационарное состояние, т.е. потерять способность к автотрансформации. Это состояние сети и будет решением задачи. Как правило, в этом случае ДАС будет состоять только из пассивных автоматов, представляющих данные. Такая модель вычислений несколько напоминает потоковую модель, однако сеть в потоковой модели не изменяется и состоит из одних операторов, через которые по связывающим их дугам перемещаются данные. ДАС состоит из разнородных автоматов (в том числе и из автоматов типа «данные») и непрерывно изменяется. Важным является то, что любое множество независимых друг от друга ДА, готовых к выполнению своей функции, может работать параллельно, а «готовность» автоматов

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

определяется только их состоянием и состоянием ближайших соседей. Типы автоматов и их функционирование будет рассмотрено несколько ниже.

3. Аппаратная реализация и свойства МДА

Реализация параллельной машины, основанной на ДАС, может быть выполнена как чисто программная, например, в сети из ПЭВМ (компьютер кластерного типа). В этом случае неизбежна интерпретация сетевой программы и свойств ДА, что приведет к низкой эффективности реализации. Чисто аппаратная реализация также затруднительна, так как возможностей динамического изменения структуры процессоров и связей между ними до недавнего времени не существовало вовсе, а в настоящее время они весьма ограничены, даже при использовании перепрограммируемых схем гибкой логики, например, выпускаемых фирмой Altera. Поэтому была предложена архитектура мультипроцессора, получившего название МДА. Краткое описание типовой структуры вычислительного модуля МДА и конфигурации связей между модулями приведено в [5].

Вычислительный модуль МДА состоит из нескольких специализированных процессоров, таких как исполнительный процессор (ИП), управляющий процессор (УП), коммутационный процессор (КП) и другие вспомогательные процессоры. Конфигурация связей между процессорами (вычислительными модулями) в общем случае не имеет значения, так как коммутационный процессор (КП) осуществляет «интеллектуальную» маршрутизацию и может отправить объект по любому свободному пути, однако, при наличии возможности, выбирает оптимальный путь. Наиболее подходящей является рекурсивная кластерная конфигурация, аналогичная «гиперкубу». В этой конфигурации четыре ВМ соединяются «каждый с каждым» в кластер, при этом остаются четыре свободных канала, при помощи которых кластеры соединяются в кластеры более высокого уровня. Такая конфигурация сети обеспечивает логарифмическую зависимость максимальной длины пути от полного количества модулей в системе. Физическая природа связей может быть произвольной, при наличии соответствующих каналов процессоры могут располагаться на значительных расстояниях, т.е. система в целом может быть распределенной. Ни общие шины, ни общая память не используются между модулями, что исключает конфликты и облегчает управление.

Структура УП, КП и других вспомогательных процессоров достаточно проста, в функции УП входит анализ состояния ДА (в дальнейшем будем называть их узлами программной сети, или просто узлами, либо объектами) и принятие решений о их дальнейшем функционировании, в том числе порождение и уничтожение объектов, а также управление памятью. ИП выполняет конкретные функции по обработке информации (например, вычисления), не затрачивая времени на управление и коммуникационные функции. КП выполняет функцию передачи объектов (программ и данных) между модулями по высокоскоростным каналам, которых имеет не менее четырех. При этом ИП, УП и КП могут быть реализованы как на стандартном микропроцессоре, так и схемах гибкой логики с возможностью перепрограммирования структуры процессора при переходе к выполнению другого объекта, либо в смешанном варианте. Дополнительно в модуль могут включаться процессор ввода-вывода (ПВВ) и контрольный процессор (КоП), при их отсутствии их функции распределяются между УП и КП. Все процессоры в ВМ работают параллельно, асинхронно и независимо, а

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

взаимодействие между ними осуществляется при помощи системы аппаратно поддерживаемых очередей, размещаемых в локальной памяти процессоров.

Конфигурация модулей и принципы работы МДА обеспечивают ряд полезных свойств. Первое – это реально высокое быстродействие, обеспечиваемое разделением функций и, следовательно, практически полной загрузкой ИП полезными вычислениями, множеством высокоскоростных каналов для передачи информации между ВМ и избыточностью путей передачи, а также равномерной загрузкой всех ВМ, которая позволяет выполнять задачи, которые обладают способностью к распараллеливанию, на максимально возможном числе процессоров, что обеспечивает для многих задач, либо для совокупностей различных одновременно выполняемых задач, практически линейную зависимость роста производительности от числа ВМ, которое может быть произвольным (оно даже не обязано быть степенью числа 2). Равномерная загрузка ВМ обеспечивается тем, что любой порождаемый объект с большой вероятностью транспортируется в произвольный ВМ, с учетом загрузки модуля и длины пути. В случае плохо распараллеливаемых задач имеется возможность запускать одновременно множество разных программ, либо много экземпляров одной программы с разными данными, что также позволяет максимально загрузить процессоры. Второе – это практическая независимость программы от количества модулей и конфигурации связей между ними, что снимает с программиста необходимость явно заботиться о распараллеливании задачи (если для этого нет особых причин). Кроме того, это дает возможность постепенного наращивания вычислительной мощности мультипроцессора без замены программного обеспечения и прикладных программ. Третье — это очень высокая надежность. Надежность аппаратуры обеспечивается подсистемой контроля (это часть УП, либо отдельный процессор — КоП), которая динамически перераспределяет работу при выходе из строя модулей, обеспечивает резервное дублирование информации и программ, непрерывно контролирует как аппаратуру, так и правильность сетевых программных структур, программная надежность обеспечивается тем, что каждый объект в программе имеет свое отдельное адресное пространство, поддерживаемое аппаратно (выделение и освобождение памяти также поддерживается аппаратно), причем любой объект (даже не программа в целом) имеет физический доступ только к своим областям памяти и к областям памяти своих ближайших соседей (последнее только в рамках разрешенных ему функций). Аналогичным образом организуется и внешняя память (например, диски), если она присоединена непосредственно к ВМ, что практически исключает возможность несанкционированного доступа к программам и информации, в том числе и распространение вирусов. Подсистема контроля также следит за чрезмерным захватом объектами ресурсов (например, в результате чрезмерного «размножения» объектов), предотвращая клинчи, выявляет и уничтожает объекты, не имеющие «хозяина», которые могут возникать в результате сбоев или неправильной работы программ. Четвертое — отсутствие необходимости синхронизации параллельно выполняемых программных объектов, поскольку они управляются состоянием данных и других объектов (однако имеется возможность явной синхронизации и привязки ко времени для тех случаев, когда это действительно необходимо, например, в задачах реального времени). Пятое свойство заключается в том, что единицей выполнения, управления и транспортировки является не программа, а отдельный узел программной сети (объект). В результате МДА не требует никаких специальных средств для обеспечения мультизадачности и многопользовательского режима

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

работы. Кроме того, это облегчает компоновку программ, позволяя собирать ее из отдельных «кирпичиков», среди которых может быть немало стандартных.

Теперь перейдем к более подробному рассмотрению программных аспектов МДА, структуры и функционирования программы.

4. Сетевое представление программы

Программа в МДА является всего лишь представлением ДАС, следовательно, это прежде всего сеть. Программная сеть имеет специальное внутреннее представление в МДА, она может быть представлена в виде графа на специальном графическом языке программирования ЯРД, некоторые свойства которого рассматриваются в последнем разделе, либо в виде текста — текстовый вариант того же языка ЯРД синтаксически напоминает несколько модифицированный Паскаль (из-за недостатка места он здесь не описывается, но приводимые примеры достаточно понятны), однако, семантика языка совершенно нетрадиционная, поскольку это всего лишь описание программной сети и, в частности, порядок выражений во многих случаях безразличен. При этом существует достаточно жесткая и однозначная связь между внутренним представлением программной сети, графическим и текстовым представлениями той же программы. Следует также иметь в виду, что программист описывает лишь начальное состояние сети, в то время как внутреннее представление непрерывно меняется в процессе выполнения программы.

Исторически язык ЯРД «вырос» из ранее разработанного языка РЯД [6] с весьма неудачным синтаксисом, сохранив его семантику. Некоторое краткое описание основных характеристик языка ЯРД представлено в [7].

Каждый узел программной сети (объект) состоит из дескриптора и тела. Упрощенный пример структуры объекта приведен на рис. 1. Структура программы в МДА изначально является объектно-ориентированной, хотя имеются определенные отличия от ставшего уже традиционным объектно-ориентированного подхода к программированию.

Тело представляет собой массив слов с линейной адресацией, начинаемой с нулевого адреса (смещения), хотя физически он может быть расположен в памяти в виде раздельных блоков стандартной длины, наподобие того, как файлы в IBM PC размещены в перемежающихся кластерах, но программе физические адреса недоступны. Тело может содержать данные, фрагменты программы (процедуры), либо ссылки на другие объекты (например, на объекты-поля, если данный объект — структурный).

Дескриптор представляет собой небольшой блок памяти (имеет размер, кратный 16 словам), именуемый «микроблоком» и содержащий всю описательную информацию данного объекта. Это класс объекта, определяющий его поведение в сети, состояние объекта (истинное, ложное, неопределенное или безразличное), характеристики тела — размер и структурные особенности, а также некоторое множество *связей*.

Связи — это указатели на части данного объекта (тело) и на другие объекты. К «другим объектам» относятся: «объект-хозяин» (ни один объект в МДА не может не иметь «хозяина», причем каждый объект «знает» также «свой номер у хозяина»), «объект-тип» (это определяет конкретные свойства данного объекта и его программы — методы), «объект-ресурс», (определяет размещение объекта в модулях, связь с внешними устройствами, а также отчасти распространение объекта при его «размножении», т.е. порождении аналогичных

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

объектов), «объекты-соседи» (эти связи называются «примитивными отношениями» и определяют отношения с соседними объектами в сети, это могут быть отношения «аргумент-результат», «часть-целое», «отношение следования», при явном указании последовательности выполнения или синхронизации, а также ряд других). Связи представляют собой ссылки различного вида, содержание и внутренняя структура которых зависит от взаимного расположения объектов (как в физическом смысле, т.е. расположение в одном или разных модулях, так и в логическом, т.е. на одном или разном уровнях иерархии), так и от стадии обработки объекта (от последовательности индексов в иерархии до физического адреса). Следует напомнить, что физические адреса недоступны программам, процедуры объекта не могут читать или изменять поля своего дескриптора иначе, чем через специальные аппаратно-поддерживаемые функции операционной системы, в которых они, как правило, оперируют ссылками типа «номер связи – индекс» или «номер аргумента – смещение в теле».

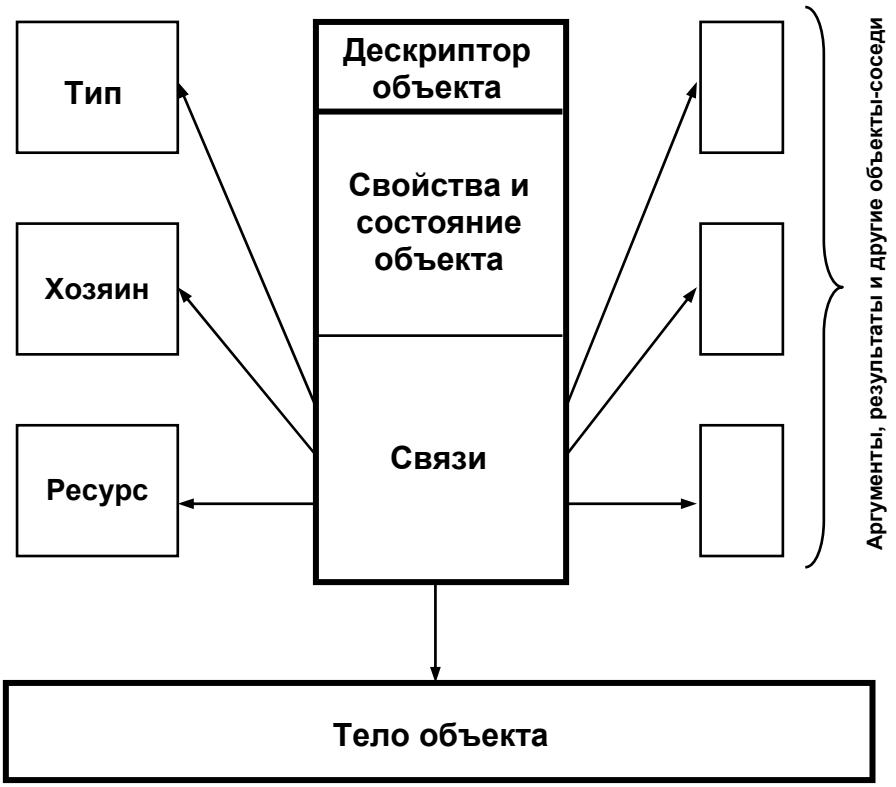

Рис. 1. Упрощенная структура объекта программной сети и его связей

Каждый объект принадлежит к одному из семи классов, которые, как уже говорилось выше, определяют его поведение в сети, а также к одному из типов внутри своего класса. Типов может быть сколько угодно, при этом типы могут определяться и программистом, но для этого он должен написать методы этого типа, отличающие его от наследуемых типов или классов, причем методы пишутся обычными средствами (язык Ассемблера или, например, C), с соблюдением определенных правил и ограничений.

Отличие между типами и классами состоит только в том, что различия классов играют важную роль в распараллеливании программы, в то время как типы определяют лишь разные варианты выполнения объектов в рамках своего

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

класса. Классы объектов следующие: *данные, операторы, ссылки, отношения, ресурсы, типы и структуры (подсети)*. В следующем разделе мы рассмотрим особенности этих классов, их поведение в программной сети и влияние на распараллеливание. На рис. 2 показаны графические обозначения объектов разных классов, используемые при графическом программировании.

5. Функционирование объектов и управление параллельным выполнением программы

Все объекты в МДА, как и в объектно-ориентированных языках программирования, составляют иерархию типов, причем корневым типом является базовый объект, в котором сосредоточены наиболее общие свойства всех узлов программной сети. От базового объекта, многие свойства которого поддерживаются на уровне аппаратуры или операционной системы, наследуют свойства семь вышеперечисленных классов, а от них уже – все множество типов. Однако, в отличие от традиционного объектно-ориентированного подхода, принадлежащие типам процедуры (методы) сосредоточены в объектах специального класса **«тип»** (**type**). Тело объекта этого класса содержит массив ссылок на методы, аналогичный таблице виртуальных методов (ТВМ или VMT) традиционных объектно-ориентированных языков. Кроме того, в объектах класса «тип» содержится информация о структуре данных соответствующего типа и о типе элементов или полей (но не о конкретных границах массивов). Сами «типы» организованы в иерархию при помощи связей типа «хозяин-раб». Каждый объект любого другого класса обязательно связан с одним из объектов класса «тип». Это позволяет повысить гибкость системы, в частности позволив объектам менять тип и даже класс. В текстовой форме языка связь с типом обозначается аналогично Паскалю, двоеточием, или ключевым словом **is** (x **:** real; M **is** matrix;). Набор методов в МДА в значительной мере регламентирован правилами поведения объектов в сети. Обязательными являются методы создания (инициализации), уничтожения и выполнения объектов, определения или изменения их состояния, структуры и связей, и ряд методов, специфичных для каждого класса. Дополнительные методы являются локальными в данном типе и не могут вызываться из методов других типов. Вызовы стандартных методов, как правило, осуществляются операционной системой в определенных ситуациях. Объекты класса «тип» могут быть общими для всех программ и в этом случае они виртуально включаются в операционную систему и распространяются во все модули системы, а также локальными в программе. Первые являются общим разделяемым программным ресурсом и никогда не уничтожаются. Вторые могут уничтожаться при завершении программы, либо в случае, когда в динамически изменяемой программной сети не осталось объектов соответствующего типа.

Любой вновь созданный объект (порожденный другим объектом либо перемещенный из окружающей среды, т.е. из другого модуля или из внешних устройств), а также объект, изменивший свое состояние, попадает в очередь к УП, который анализирует его состояние и состояние его соседей по сети и, в соответствии с их классами и состояниями, принимает решение о дальнейшей судьбе объекта, который может быть отправлен на выполнение в данный или в другой модуль (в этом случае он ставится в очередь к ИП данного модуля или в одну из очередей КП, соответствующих другим модулям или кластерам), может породить другие объекты или быть уничтоженным. Если объект или его бли-

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

жайшее окружение не готовы ни к одному из этих действий, то они «засыпают» до того момента, пока снова не изменится состояние этого объекта или одного из соседей. Постановка объекта в очередь к КП осуществляется в том случае, если он связан в программе с определенным ресурсом (модулем), если его «сосед», готовый к исполнению, находится в другом модуле, либо если очередь к ИП данного модуля заполнена. Поскольку размер очереди ограничен, то большая часть объектов будет отправляться для исполнения в другие ВМ. Это лишь один из механизмов, который приводит к автоматическому распределению фрагментов программы между модулями.

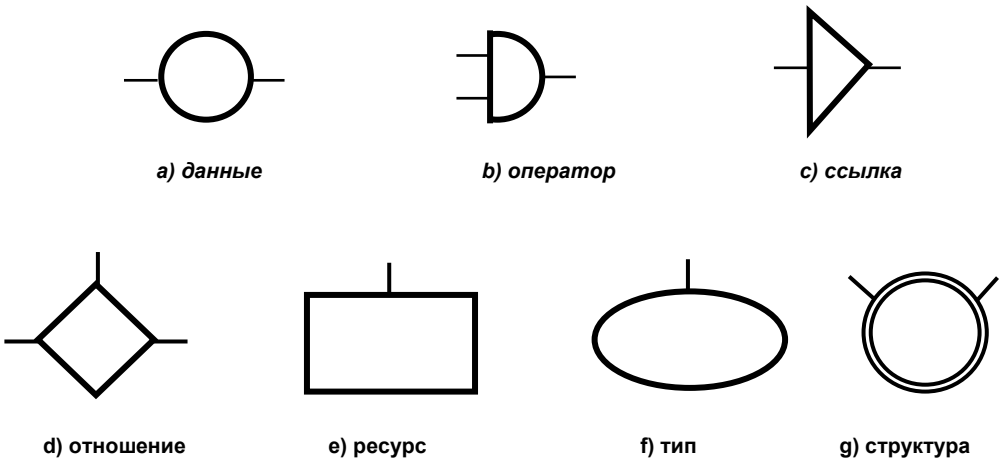

Рис. 2. Графические обозначения объектов различных классов в языке ЯРД.

Объекты класса **«данные»** (**data**) могут содержать информацию произвольного типа и структуры, которая содержится в теле объекта. Наиболее типичным является одномерный массив, но возможен двумерный или многомерный массив, организованный как массив ссылок на другие массивы, которые могут быть как объектами класса «данные», так и совокупностью блоков памяти, организованных каким-либо другим способом. Возможны также структуры или массивы структур, а также области памяти, организованные произвольным образом. Одиночные данные, как правило, не выделяются в отдельные объекты (это не выгодно с точки зрения расхода памяти), а группируются в структурные области памяти. В тех случаях, когда выделение одиночной переменной или константы является необходимым, ее значение располагается прямо в дескрипторе, т.е. тело объекта отсутствует. Методы соответствующего типа, в основном, осуществляют преобразования данных к другому типу или изменение их структуры, в том числе и размера или размерности массивов. Объект класса «данные» (его дескриптор) первоначально находится в неопределенном состоянии. Когда тело объекта заполняется данными в результате работы некоторого оператора, либо в результате их ввода из окружающей среды или из другого модуля, объект переходит в истинное состояние (состояние готовности). При возникновении ошибок в процессе вычислений или передачи объект переходит в ложное состояние. Состояние готовности данных может служить основанием для активизации (исполнения) других объектов, например, операторов, для которых данный объект является аргументом, либо для изменения состояния структурного объекта, частью которого он является. Если объект-аргумент полностью использован всеми связанными с ним операторами, то он может быть уничтожен, либо, если он имеет специальный признак «кратности»,

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

т.е. длительности существования, то он снова переводится в неопределенное состояние. Как правило, готовность объекта означает готовность всех его элементов, но может быть организована и «частичная готовность», соответствующая готовности группы компонентов структуры, если это позволяет работать соответствующему оператору. Например, для обработки сигналов и других непрерывных процессов используется специальный тип **«поток»** (**stream**), который представляет собой потенциально бесконечный массив, описание которого содержит границы «окна», т.е. некоторого блока памяти заданного размера, достаточного для обработки. В этом случае готовность наступает при заполнении «окна», а тело объекта представляет собой последовательность «окон», которые поступают на обработку по заполнении и уничтожаются после окончания обработки. Сам объект при этом не уничтожается.

Объект класса **«оператор»** (**operator**) выполняет некоторую обработку информации (вычисления), содержащейся в объектах-аргументах и заполняет тело объекта-результата. Тело соответствующего объекта-типа содержит метод (методы), необходимые для выполнения соответствующих вычислений, а тело самого объекта-оператора представляет собой область памяти, в которой располагаются локальные данные, стек и другая информация, необходимая для работы оператора. Готовность объекта определяется только наличием в данном ВМ дескриптора оператора и соответствующего типа (вместе с телом). Поэтому стандартные операторы готовы всегда, а локальные в программе – при поступлении в модуль объекта-типа. Оператор активизируется и отправляется на исполнение в случае готовности его аргументов и неопределенного состояния результата. Однако, существует ситуация, когда готовый в исполнению оператор не выполняет вычислений. Это происходит, если тип его формальных аргументов не соответствует фактическим по структуре. Например, оператор предназначен для обработки векторов, а фактические аргументы – матрицы. В этом случае включается механизм порождения множества подсетей из аналогичных операторов, аргументами которых являются строки (или столбцы) матриц-аргументов. Разумеется, механизм порождения встроен в специальный метод данного оператора. Поскольку число порождаемых объектов может быть велико, очередь к ИП быстро переполняется, и остальные объекты распространяются по сети. Так работает еще один механизм распараллеливания, хотя он легко реализуется только в простых случаях, таких как, например, сложение матриц. Более сложные случаи легко описываются при помощи ссылок. Выполнивший свою функцию оператор, как правило, уничтожается, однако, если оператор породил свои копии, то он будет уничтожен только после уничтожения всех потомков. Конечно, и «расчлененный» на строки аргумент также будет уничтожен только после уничтожения всех строк. Существует особый случай, когда оператор не уничтожается после окончания работы, а изменяет свой тип (и класс) на тип результата. Это происходит, если выход оператора присоединен ко входу другого оператора в качестве аргумента. В языке это изображается функциональной записью, например, вида

Y := k * sin (x);

здесь оператор **sin** является аргументом оператора умножения. В этом случае оператор имеет тело, размер и структура которого определяется типом результата, в котором и накапливается результат.

Объект класса **«ссылка»** (**reference**) является важнейшим в МДА с точки зрения влияния на распараллеливание вычислительного процесса. В отличие от традиционного понимания ссылки как указателя на некоторый другой объект,

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

в МДА ссылка — это некоторый метод доступа не только к другому объекту, но также к любой его части, причем способ выделения частей объекта может быть произвольной сложности. Сюда включаются следующие основные типы ссылок: *простые указатели* — ссылки, обеспечивающие косвенный доступ к данным, в отличие от традиционных, они позволяют ссылаться «вверх», т.е. на «хозяина», а также на его компоненты; *сечения, вырезки и выборки* – регулярный способ выделения частей массивов или структур; *сборки* - способ объединения полей или подмассивов различных структурных объектов; *функциональные ссылки* — позволяют организовывать доступ к данным по сложным критериям (вычисляемые вырезки, сопоставление с образцом и т.п.); *транзитные ссылки* — обеспечивают удаленный доступ (доступ к данным в других ресурсах). Тело ссылки содержит информацию для осуществления соответствующего доступа (например, диапазоны индексов по каждому измерению массива или множества индексов или номеров полей, или образец для поиска и сопоставления информации), а тело соответствующего типа — процедуры, обеспечивающие соответствующие структурные преобразования. Но выборками и вырезками, пусть даже и очень сложными, функция ссылок не ограничивается. Во-первых, при определении готовности некоторого оператора, аргументом которого является ссылка, готовность может быть определена при частичной готовности данных (достаточно, чтобы готовыми были только те данные, на которые указывает ссылка, а не весь массив), что ускоряет принятие решения и отправку оператора на выполнение. Во-вторых, наличие ссылки в качестве аргумента заставляет систему порождать соответствующие группы объектов, которые могут выполняться параллельно и на разных ВМ. Это хорошо видно на примере программы умножения матриц, который рассмотрен в следующем разделе. Наличие ссылок вида **a [i,]** и **b [, j]** (они обозначают, соответственно, строки первой матрицы и столбцы второй) приводит к порождению множества подсетей, которые выполняют скалярные произведения векторов (строк и столбцов), причем эти подсети могут выполняться на разных модулях одновременно. Ссылка **c [i, j]** автоматически определяется как сборка, поскольку она является результатом, а не аргументом, и ее выполнение состоит в объединении отдельных элементов матрицы-произведения в единый результат операции умножения матриц. С ссылками (а также с ресурсами) связано также важное понятие — кванторы. Используется два типа кванторов — **all** (все) и **any** (любой). Первый означает, что некоторое действие выполняется для всех объектов или элементов объектов заданного типа, а второй относит действие к любому произвольному элементу из множества. В простых случаях кванторы явно не обозначаются, а подразумеваются. Например, полная форма записи ссылки **a [i,]** имеет вид **a [any i, all j]**. Ссылки в данном случае могут быть и более сложными, например, в случае матриц очень большого размера можно задать разделение не только по строкам и столбцам, но и на более мелкие части, задав диапазоны индексов (например, **a [k*n..k*n+n-1]**), что позволяет задавать распараллеливание более мелких фрагментов вычислений. Данный пример является регулярным для наилучшего понимания, однако этот механизм распараллеливания будет так же хорошо работать, если вместо матриц выступают сложные структуры данных, ссылки указывают на поля или совокупности полей записей, к тому же включают в себя и сопоставление с образцом (например, поиск в распределенной базе данных), а оператор выполняет более сложные действия, чем скалярное произведение векторов.

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

Объект класса **«отношение»** (**relation**) аналогичен оператору, но, в отличие от последнего, активизируется в случае любого изменения состояния любого из своих аргументов. Кроме того, тип отношения может включать в себя методы, которые способны вычислять любой из аргументов в зависимости от других при изменении последних таким образом, чтобы заданное отношение всегда выполнялось. Такое свойство отношения может быть использовано, например, для моделирования различных процессов. Например, при моделировании электрических цепей можно использовать два отношения, которые обеспечивают выполнение законов Кирхгофа, т.е. одно отношение «отслеживает» нулевую сумму токов в узлах электрической схемы, а второе – разности потенциалов по контуру. Малейшее приращение одного из токов или потенциалов в моделируемой цепи приведет к активизации соответствующего отношения и вычислению новых значений токов и потенциалов в остальных частях цепи.

Объект класса **«ресурс»** (**resource**) является «представителем» физических ресурсов МДА и позволяет определять ввод, вывод или размещение остальных объектов программы в различных устройствах компьютера. К таким устройствам относятся, прежде всего, модули, внешняя память (диски), а также различные устройства ввода-вывода, в том числе и специализированные, такие как источники радарных или гидроакустических сигналов, датчики и органы управления систем управления или робототехнических систем и другие. К более сложным ресурсам могут относиться, например, серверы распределенных баз данных, сайты Интернета и т.п. Основным свойством ресурса является способность перемещать или размещать данные или программы. Если некоторый объект связан с ресурсом типа «ввод» (например, **input** или **stdin**), то это автоматически приводит к вводу данных или программ с соответствующего устройства, после чего объект приобретает состояние готовности. Ресурс типа «вывод» (например, **output** или **stdout**) обеспечивает вывод на некоторое устройство. Связь с ресурсом типа «модуль» обеспечивает размещение соответствующего объекта в указанном модуле или во всех модулях из указанного диапазона, или в одном (произвольном) модуле. Методы ресурса представляют собой драйверы периферийных устройств, либо процедуры, которые ставят объект в очередь к КП для его перемещения в указанные модули или устройства. При этом могут осуществляться различные преобразования форматов, типов или структур данных. К ресурсам, как и к ссылкам могут быть применены кванторы **all** или **any**. Применение квантора **all** к некоторому объекту приводит к его копированию во все устройства (модули) из указанного диапазона (например, **all units (1..N)**), либо во все модули, если диапазон не указан. Применение квантора **any** приводит к перемещению объекта в любой произвольный модуль. По умолчанию программа в целом, как структурный объект, связана с ресурсом типа «все модули», а ее компоненты – с ресурсом типа **any**. Таким образом, объекты-ресурсы, с одной стороны, существенно облегчают программирование операций обмена (достаточно связать объект с соответствующим ресурсом, чтобы обеспечить его ввод или вывод со всеми необходимыми преобразованиями), а, с другой стороны, являются простым и эффективным средством влияния на процесс распараллеливания задачи. В то же время, если для какого-то объекта конкретный ресурс явно не указан в программе, то система может выбрать ресурс из имеющихся свободных ресурсов подходящего типа. Ресурс, как и другие объекты, может быть аргументом некоторого другого объекта, в том числе и структурного объекта, представляющего программу в целом. Это позволяет использовать программу или любую ее часть по-разному, присоеди-

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

няя ее к различным источникам и приемникам информации. В текстовом представлении программы связь объекта с ресурсом обозначается при помощи слова **at** или символа **“@”** (например, X **at input**; Y **@ output**; Z **at all**).

Объект класса **«структура»** (**structure**) представляет собой функционально законченный фрагмент сетевой программы (подсеть). Тело объекта содержит массив ссылок на объекты, составляющие данную подсеть, среди которых могут быть объекты любых классов. Для этих объектов структурный объект является «хозяином». Следует отметить, что сложные структуры данных являются все же объектами класса «данные», а не класса «структура», поскольку они однородны по классу компонентов, а их связи с компонентами являются связями «часть-целое». Готовность структурного объекта определяется наличием в данном модуле всех его компонентов, а не их готовностью. Исполнение структурного объекта состоит в исполнении всех его компонентов (точнее, в постановке их в очередь к УП для анализа и возможного последующего исполнения). Уничтожается структурный объект только после уничтожения всех его компонентов. Структурный объект введен только с целью улучшения модульной структуры программы и удобства ее построения, он может быть единицей компиляции, хранения и пересылки. Кстати, программа в целом также является структурным объектом специального типа.

Таким образом, распараллеливание программы в МДА осуществляется автоматически благодаря свойствам и поведению самих объектов, составляющих программу. Программист может либо вообще не заботиться о распараллеливании, либо влиять на него, если необходимо, задавая в программе соответствующие сложные ссылки, связи с ресурсами, элементы синхронизации или привязки к временным интервалам. Операционная система МДА очень проста, часть ее функций, таких как распределение памяти или запуск объектов на выполнение выполняются на аппаратном уровне, а интерфейсные функции (взаимодействие с пользователями, утилиты и т.п.) выполняется такими же сетевыми программами, как и прикладные, единственное отличие заключается в том, что программы операционной системы имеют приоритет, позволяющий им работать с защищенными областями памяти. Ядро операционной системы может быть реализовано как набор методов базовых типов (в первую очередь — ресурсов), поэтому многие функции операционной системы являются просто свойствами объектов программы, наследуемыми от базовых типов и классов.

6. Графическое сетевое программирование

Поскольку программа в МДА представляется в виде сети, то естественным образом напрашивается возможность «рисования» программы в виде некоторого графа. Поэтому, язык программирования ЯРД — прежде всего графический язык. Программа создается при помощи специального графического редактора, который позволяет размещать обозначения объектов и соединять их связями разных типов. Внутри самого объекта помещается его индивидуальное обозначение, которое может быть идентификатором, пиктограммой, либо более сложным обозначением (например, индексное выражение для ссылки). Поскольку разместить в обозначении объекта много информации невозможно, то объект можно «раскрыть» подобно тому, как это делается в современных визуальных системах программирования, и задать различные свойства объекта, а для объектов-типов – написать соответствующие процедуры-методы. Раскрытие же структурных объектов позволяет задавать их сетевые представления

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

отдельно, что обеспечивает модульность программы. Однако, этот метод существенно отличается от существующих визуальных методов, поскольку там визуально создаются лишь экранные формы, а здесь рисуется сама сетевая программа. На рис. 3 показан пример очень простой программы, которая выполняет умножение двух квадратных матриц произвольного размера.

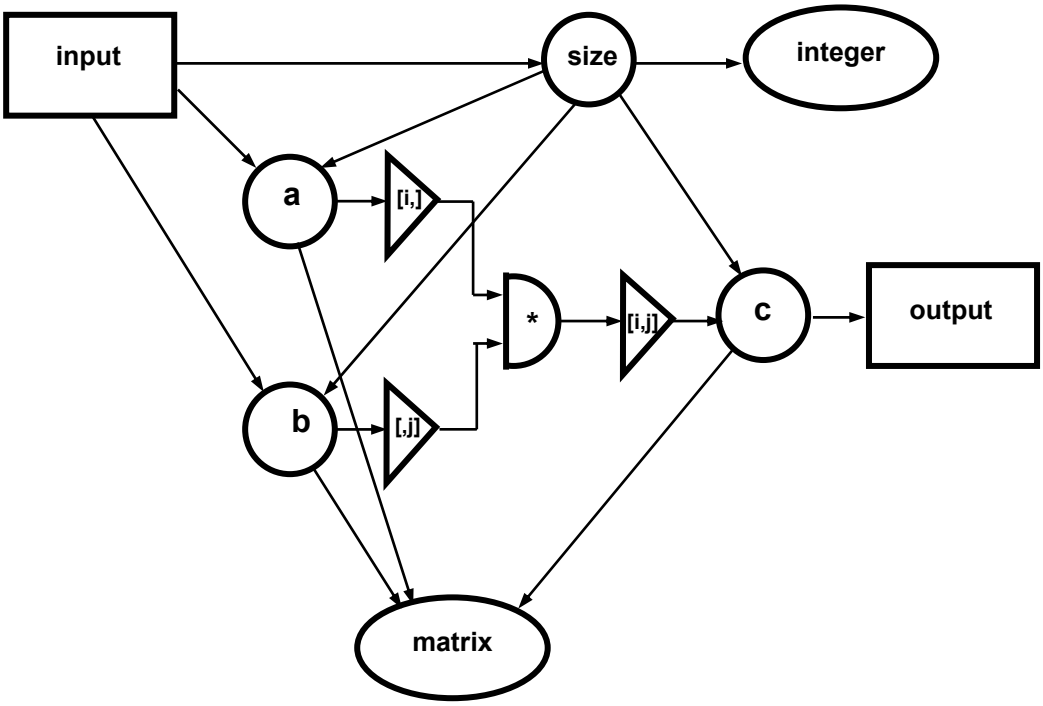

Рис. 3. Пример графического представления программы на языке ЯРД

В текстовой форме языка ЯРД эта же программа может иметь примерно такой вид (возможны различные варианты записи):

```
program MultMatr (input, output);
size : integer at input;
a, b : matrix [size, size] at input;
c : matrix [size, size] at output;
begin
  for all i, j do
  c [any i, any j] := a [i,] * b [, j];
end.
```

Следует отметить, что оператор **for** не является оператором цикла, а, скорее, является указанием, что все элементы матрицы-результата нужно вычислять параллельно.

На рис. 4 показано, каким образом происходит порождение новых фрагментов программы (подсетей) при автоматическом распараллеливании. Порожденные объекты сохраняют связь с объектами, породившими их (на рисунке показаны не все связи), что позволяет в дальнейшем собирать структурные результаты или массивы из отдельных элементов.

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

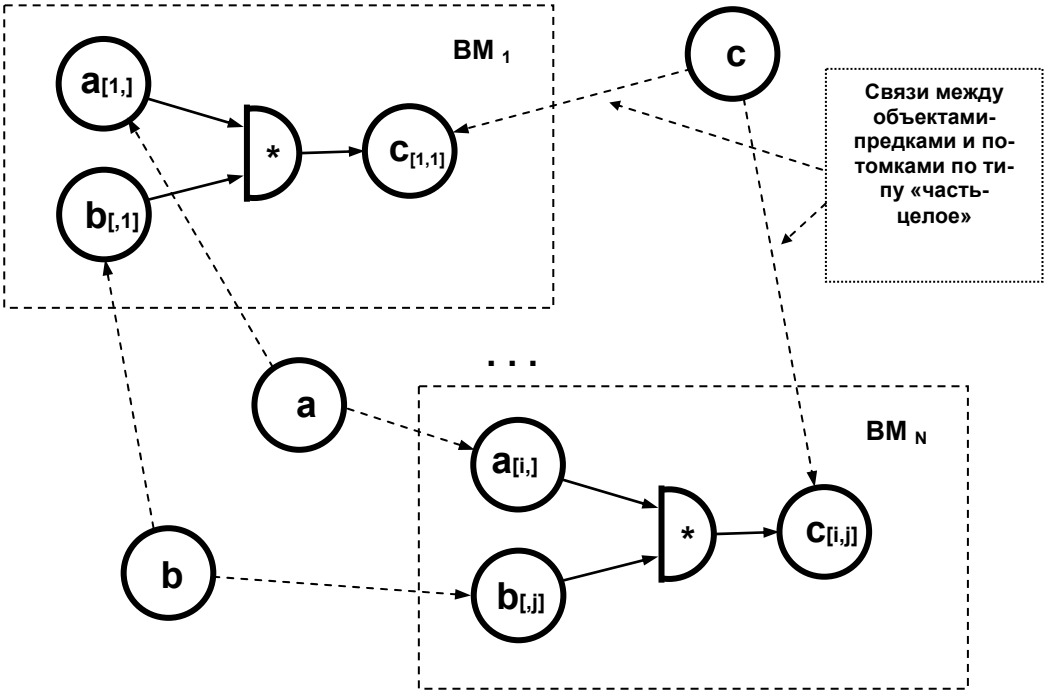

Рис. 4. Пример порождения групп объектов при автоматическом распараллеливании фрагмента программы

7. Заключение

В заключение покажем, почему МДА и соответствующий метод программирования параллельных вычислений являются ориентированными на решение конкретной задачи, а не на преобразование задачи под машину. Как уже было отмечено, практически любые задачи легко представляются в сетевой форме. Архитектура МДА ориентирована как раз на такое представление. В процессе выполнения сетевая структура программы изменяется. Но в каждый конкретный момент времени в машине присутствуют объекты (узлы этой сети), причем аппаратура подстраивается под структуру программы, поскольку в памяти имеются только объекты одной или нескольких программ, а в момент выполнения процессоры поддерживают структуру и связи этих объектов, т.е., в конечном счете, структуру задачи. Реализованный на схемах гибкой логики процессор может динамически менять свою структуру и набор команд, что сильно понижает требования к объему аппаратуры. Например, в момент вычисления скалярного произведения строк и столбцов матриц процессор может «не уметь» делать ничего другого, но это произведение он может выполнять наиболее эффективным способом. Это позволяет в том же объеме аппаратуры реализовать не один, а множество специализированных «под задачу» процессоров, что существенно повышает общую эффективность системы. В то же время, такая структура выполняет распараллеливание естественным для задачи и наиболее эффективным способом, выполняя параллельно обработку всех объектов программы, которые к этому готовы.

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

Литература

[1] *Glushkov V. M., Ignatyev M. B., Myasnicov V. A., Torgashev V. A.* Recursive Machines and Computing Technology // IFIP Conf. Proc. — Amsterdam: North-Holland Publ. Co., 1974, — p. 65–70.

[2] *Торгашев В. А.* Управление вычислительными процессами и машины с динамической архитектурой. В сб.: Вычислительные системы и методы исследований и управления автоматизации. — М.: Наука, 1982, — сс. 172–187.

[3] *Torgashev V. A. and Plyusnin V. U.* Dynamic Architecture Computers // Proc. of The Int. Conf. Parallel Computing Technologies. ReSCo, — Moscow: 1993, — p. 25-29.

[4] *Торгашев В. А.* Машины с динамической архитектурой // Теоретические основы и прикладные задачи интеллектуальных информационных технологий. Российская академия наук, Санкт-петербургский институт информатики и автоматизации, — СПб: 1998, — сс. 154–165.

[5] *Мыскин А. В. , Торгашев В. А. , Царев И.В.* Процессоры с динамической архитектурой на основе схем гибкой логики // Труды СПИИРАН. Вып. 1, т. 1, — СПб: СПИИРАН, 2002.

[6] *Торгашев В. А.* РЯД — язык программирования для распределенных вычислений. Препринт №28. Академия наук СССР, Ленинградский научно-исследовательский вычислительный центр, — Л.: 1984, — сс. 3–48.

[7] Torgashev V. A., Tsaryov I. V. A Parallel Programming Language for Dynamic Architecture Computers // Proc. of The Int. Conf. Parallel Computing Technologies. ReSCo, — Moscow: 1993, — p. 31–34.