УДК 681.3.06:681.32

СРЕДСТВА ОРГАНИЗАЦИИ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ И ПРОГРАММИРОВАНИЯ В МУЛЬТИПРОЦЕССОРАХ С ДИНАМИЧЕСКОЙ АРХИТЕКТУРОЙ

В. А. Торгашёв, И. В. Царёв

Аннотация

В статье рассматриваются методы и средства организации программ в мультипроцессорной среде с динамической архитектурой как автотрансформирующихся сетей, состоящих из объектов различных классов (данных, операторов, ссылок и т.п.). Предлагается объектно-ориентированный метод программирования автотрансформирующихся сетевых программ, отражающих в большей степени структуру решаемой задачи, чем свойства вычислительной среды. Рассматриваются средства поддержки данного подхода на аппаратном уровне и уровне операционной системы. Обсуждается метод графического проектирования параллельно выполняемых программ в сетевом представлении.

1. Введение

Идея параллельного выполнения разных частей программы на различных процессорах, возникшая сравнительно давно, тем не менее до настоящего времени не нашла полностью удовлетворительного решения. Существующие мультипроцессорные компьютеры показывают быстродействие, приближающееся к «пиковому», т.е. теоретически максимальному, только на узком классе задач, зависящем от конкретной архитектуры компьютера, на многих других задачах реальное быстродействие оказывается на порядок или два ниже пикового. Для этого существует ряд причин. Оставляя в стороне чисто технические причины, такие как низкая пропускная способность каналов, конфликты, возникающие при использовании общих ресурсов, например, памяти, и другие, отметим, что одна из основных причин – «семантический разрыв» между между структурой задач, алгоритмов и архитектурой компьютеров, приводящий к тому, что разработка параллельных программ требует значительных усилий от разработчиков, поскольку средства параллельного программирования сводятся, в основном, к расширению традиционных последовательных языков программирования процедурами порождения, уничтожения и синхронизации параллельных процессов и обмена сообщениями, причем управлять этими сложными действиями должен программист, что требует огромной квалификации и опыта. Но даже при их наличии не всегда удается найти удачное решение для сложных задач.

В данной статье рассматривается организация параллельных вычислительных процессов и их программирования в мультипроцессорах с динамической архитектурой (МДА), основанных на модели вычислений, именуемой «динамические автоматные сети» (ДАС), которая разрабатывается более двадцати лет в Санкт-Петербургском институте информатики и автоматизации РАН (СПИИРАН) под руководством проф. В.А. Торгашева.

2. Динамические автоматные сети

Общепринятый алгоритмический подход к решению задач, основан-

ный на идеях Тьюринга и фон-Неймана, предполагает изначально последовательный характер выполнения программы, которую затем, при наличии множества вычислительных ресурсов, приходится искусственно принудительно «распараллеливать». Существующие модели вычислений, предполагающие параллельное выполнение, такие, как потоковая модель и другие, ограничивают не только класс решаемых задач, но и их размер, поскольку являются статическими. Предположим, что требуется выполнить умножение двух матриц размером N * N. Полностью параллельная программа содержит N^3 операторов умножения, которые могут выполняться параллельно. Однако при достаточно большом значении N (например, N = 10000) в потоковой модели требуется сформировать 10^{12} параллельных операторов и такая программа окажется практически нереализуемой. Один из возможных выходов заключается в том, чтобы не создавать N^3 операторов, а порождать их динамически по мере освобождения ресурсов машины.

Более 20 лет назад В.А. Торгашевым была выдвинута идея рекурсивных вычислительных машин (РВМ) [1]. Анализ недостатков РВМ, реализованной в виде макетного образца, и дальнейшее развитие идеи привело в начале 80-х годов к созданию теории динамических автоматных сетей (ДАС) [2, 3, 4], которая была положена в основу МДА. Кратко суть этой идеи заключается в следующем. Имеется некоторое множество **A** динамических автоматов (ДА). Любой ДА $\mathbf{a \in A}$ представляет собой конечный автомат

$$\mathbf{a = \{X, Y, Q, \delta, \lambda, C, \sigma\},}$$

дополненный двумя компонентами: **C** – множество связей ДА с другими ДА и $\boldsymbol{\sigma}$ – функция связей автомата, определяющая изменение его связей в зависимости от состояния. Любой ДА $\mathbf{a \in A}$ характеризуется множеством входных каналов $\mathbf{H_a}$, каждому входному каналу $\mathbf{h_a}$ соответствует множество возможных состояний $\mathbf{x\ (h_a)}$, множеством выходных каналов $\mathbf{Z_a}$ с множеством состояний $\mathbf{y\ (q_a)}$ и множеством элементов памяти $\mathbf{S_a}$, определяющих множество внутренних состояний $\mathbf{q\ (s_a)}$. Множества входных состояний автомата **Xa** (входной алфавит), выходных состояний **Ya** (выходной алфавит) и внутренних состояний **Qa** можно определить как

$$\mathbf{X_a = \prod_{h_a \in H_a} x\ (h_a),} \quad \mathbf{Y_a = \prod_{z_a \in Z_a} y\ (q_a)} \quad \text{и}$$

$$\mathbf{Q_a = \prod_{s_a \in S_a} q\ (s_a).}$$

Определим множество связей как

$$\mathbf{C_a = \prod_{h_a \in H_a} (h_a * (Z/Z_a)) * \prod_{z_a \in Z_a} (q_a * (H/H_a)),}$$

где $\mathbf{H = \bigcup_{a \in A} Ha \cup \varnothing}$ и $\mathbf{Z = \bigcup_{a \in A} Za \cup \varnothing,}$

$\varnothing$ - пустое множество или пустой элемент множества.

Каждый элемент множества C можно представить в виде матрицы связей, число строк которого равно числу входных и выходных каналов, а число столбцов равно двум, причем левый столбец соответствует каналам автомата **a**, а правый – каналам тех автоматов из множества **A** (называемых смежными), которые связаны с данным автоматом. Если правый элемент какой-либо строки матрицы связей является пустым, то канал называется свободным, в противном случае – связанным (входные каналы ДА могут быть связаны только с выходными каналами других ДА и наоборот). ДА называется свободным, если все его каналы свободны, при этом любой свободный ДА находится в некотором начальном состоянии, общем для всех ДА.

Пусть t – дискретное автоматное время, тогда функции переходов, выходов и связей ДА можно определить с помощью рекуррентных соотношений:

$$\mathbf{Q_a\ (t+1) = \delta_a\ (Q_a(t), X_a(t))},$$

$$\mathbf{Y_a\ (t+1) = \lambda_a\ (Q_a(t))},$$

$$\mathbf{C_a\ (t+1) = \sigma_a\ (C_a(t), Q_a(t), X_a(t), X_a(t+1))}.$$

Функция связей определяет изменения связей автомата с другими, которые могут происходить на каждом такте. Сюда включается соединение некоторого свободного входа или выхода ДА со свободным выходом или входом другого ДА, если этот другой ДА – свободный, то такой процесс называется

«порождением ДА», разрыв некоторой связи с другим ДА, если при этом один из двух ДА становится свободным, то это называется «уничтожением ДА» и переключение связи со смежного ДА на смежный с ним, в этом случае смежный ДА также может стать свободным.

Область определения функции связей σ_a, в отличие от λ_a и δ_a, является бесконечной в силу бесконечности множества **A**, однако, за счет некоторых ограничений, несущественных для функционирования модели, ее можно сделать конечной. Рассмотрим последовательность ДА $a_0, a_1, \ldots, a_n$, принадлежащих множеству **A**, такую, что любые два автомата a_{i-1} и a_i, где $i = 1..n$, являются смежными. В этом случае между ДА a_0 и a_n имеется связь длиной **n**. Между любыми двумя автоматами из множества **A** может существовать некоторое множество связей. Минимальную длину связи назовем расстоянием между автоматами.

d-окрестностью автомата $a \in A$ назовем такое множество $Da \subset A$, каждый элемент которого находится на расстоянии не более **d** от **a**. Будем считать, что функция связей σ_a автомата **a** определена лишь на d-окрестности этого автомата, т.е. область определения конечна. Поскольку в каждом такте связи могут изменяться, то очевидно, что ограничение возможности изменения связей автомата d-окрестностью относится только к одному моменту автоматного времени. Если же процесс развернуть во времени, то можно считать, что за интервал времени Δt связи автомата **a** изменяются в пределах g-окрестности, где $g = (d-1)\Delta t$. Поэтому при $d \geq 2$ для любых двух автоматов **a** и **b**, расстояние между которыми в момент времени **t1** не превышает **g**, можно найти такое $t2 > t1$, при котором **a** и **b** могут быть сделаны смежными.

ДАС – это такое подмножество множества **A**, любые два автомата которого связаны между собой. Таким образом, мнодество **A** состоит из одной или нескольких ДАС и множества свободных автоматов. Будем считать, что функция связей любого ДА из множества **A** задает связи не только в d-окрестности этого ДА, но и связи со свободными автоматами. Если множество свободных автоматов бесконечно, то любая ДАС может неограниченно расти за счет включения в свой состав свободных автоматов. Таким образом функция связей любого автомата, входящего в ДАС, позволяет не только изменять конфигурацию сети, он и обеспечивает рост ДАС путем порождения ДА, включаемых в состав сети, либо уменьшать сеть за счет уничтожения автоматов.

При установлении или изменений связей между ДА автомат, устанавливающий связь (источник связи) должен получать информацию о состоянии соответствующего канала другого ДА (приемника связи) и сообщать последнему свое состояние, как и сам факт установления связи. Очевидно, что для этого связи должны быть двунаправленными, т.е. и у источника, и у приемника связи должны использоваться, по меньшей мере, один входной и один выходной каналы. Использование двунаправленных связей между ДА позволяет отказаться от общего для всего множества ДА времени и считать, что каждый ДА имеет свои собственные часы, не синхронизированные с часами других ДА, но при этом связи не равнозначны. Одна из них используется для передачи основной информации от источника к приемнику, другая же – только для передачи служебной информации о состоянии приемника источнику основной информации. Соответственно каналы любого автомата можно разделить на информационные и управляющие, число которых одинаково.

Любую ДАС можно рассматривать как виртуальную параллельную машину, если каждому автомату ДАС сопоставить некоторый программный элемент (ПЭ), причем под ПЭ, так же, как и в машине Тьюринга, может пониматься как некоторая часть программы, так и часть данных, т.е. программа не делится на две разнородные части, как это делается, например, в потоковой модели (операторная сеть и потоки данных), а рассматривается как единое целое. Любая виртуальная машина отражает наиболее существенные с точки

зрения программирования черты соответствующей реальной машины, т.е. определяет архитектуру этой машины. Поскольку структура виртуальной машины, основанной на ДАС, представляет собой сетевую структуру, динамически изменяющуюся в процессе ее функционирования, причем элементы этой структуры (ДА) имеют независимое аинхронное автоматное время, то и реализованная на этой основе реальная машина будет иметь сетевую динамическую архитектуру и будет иметь полностью асинхронное управление.

Рассмотрим взаимодействие между ДА в процессе функционирования ДАС. Без нарушения общности можно считать, что часть ДА имеет все информационные входы свободными. Такой автомат можно рассматривать как генератор входной цепочки символов. Назовем его «автоматом ввода». Аналогичным образом можно определить и автоматы вывода. Пусть имеется некоторая начальная ДАС, состоящая из автомата $\mathbf{a_0}$ и автомата ввода $\mathbf{a_i}$ ($\mathbf{a_0}$, $\mathbf{a_i \in A}$). В ДА $\mathbf{a_0}$ из $\mathbf{a_i}$ поступает линейный текст (входная цепочка), который, благодаря наличию специальных символов или их последовательностей, интерпретируется автоматом как сетевая (древовидная) структура. Каждое слово вводимого текста соответствует некоторому ПЭ, среди которых есть как терминальные (не содержашие других ПЭ) и, соответственно, нетерминальные, содержащие ПЭ более низких уровней. Между ПЭ существуют как многоуровневые связи, отражающие древовидную структуру программы, так и одноуровневые связи, характеризующие ее сетевые свойства. Каждому ПЭ входной цепочки соответствует некоторый ДА в ДАС, который порождается автоматом, принявшим данный ПЭ, при этом связь принявшего символ автомата с ДА ввода передается порожденному автомату. Если первый автомат не имел других связей, то он становится свободным (уничтожается). Если этот ПЭ – нетерминальный, то он порождает ДА, находящийся на более низком уровне иерархии, образуя таким образом иерархическую сетевую структуру. В процессе функционирования ДАС непрерывно изменяет свою конфигурацию, пока не закончится входная цепочка символов, т.е. программа, включающая в себя и данные, а также продолжает изменяться и далее, в силу свойств самих автоматов, поскольку структура вбирает в себя информацию из входной цепочки, постепенно видоизменяя ее, это называется *автотрансформацией сети*. Как правило, сначала ДАС растет, поскольку ввод входной цепочки приводит, в основном, к порождению автоматов, а в конце, по мере выполнения автоматами своих функций, - уменьшается. В какой-то момент изменения ДАС прекращаются, сеть переходит в стационарное состояние, которое и является решением задачи. В зависимости от набора функций связи ДА, может быть и такой вариант, когда программа не переходит в стационарное состояние, а начинает трансформироваться циклически, генерируя результаты в виде отдельных стационарных ДАС.

При программно-аппаратной реализации ДАС, которая не может быть бесконечной, как множество ДА, необходимо ввести понятие ресурса, в качестве которого может выступать как некоторое пространство в памяти, так и некоторый процессор. В этом случае порождение новых элементов программной сетевой структуры, соответствующей порождаемому ДА, выполняется только в том случае, если имеется в наличии соответствующий свободный ресурс, а в противном случае задерживается до освобождения такого ресурса. Это позволяет выполнять и программу решения такой задачи, которая порождает большое количество элементов, например, вышеупомянутой задачи перемножения больших матриц, при этом не требуется генерировать одновременно 10^{12} операторов.

Теперь перейдем к рассмотрению аппаратной и программной реализации данной модели.

3. Аппаратная реализация и свойства МДА

Реализация параллельной машины, основанной на ДАС, может быть выполнена как чисто программная, на-

пример, в сети из ПЭВМ. 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, 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.

В 1986-1988 гг. В СПИИРАН совместно с НИЦЭВТ был создан образец МДА ЕС-2704 на элементной базе ЕС ЭВМ, который прошел испытания, подтвердившие основные свойства МДА. Он представлял из себя мультипроцессор, состоящий из 24-х модулей и занимавший одну стандартную стойку ЕС ЭВМ, при этом показывал быстродействие около 100 млн. операций в сек. Позднее предпринимались попытки реализации МДА на более современной элементной базе (микропроцессорах серии TMS фирмы Texas Instruments). В последние годы появление схем гибкой логики позволило создать в качестве модуля МДА нового типа «процессор с динамической архитектурой» (ПДА), особенностью которого является возможность перепрограммирования схемы и функций процессора в процессе решения задачи, что несколько приблизило «идеальную» реализацию МДА. Пока этот процесс сдерживается тем, что скорость перепрограммирования гибкой логики невелика, однако улучшение их характеристик происходит довольно быстро.

Что же представляет собой аппаратная реализация МДА? Следует рассмотреть ее достаточно подробно, поскольку в МДА программная и аппаратная реализации очень тесно связаны и, в общем случае, взаимозаменяемы. МДА – это мультипроцессор, представляющий собой множество вычислительных модулей (ВМ), соединенных в сеть. Каждый ВМ представляет собой мультипроцессорную систему, в которой каждый процессор имеет свою специализированную функцию. Конфигурация модуля, количество и типы процессоров могут быть различными, но наиболее типичная (хотя и несколько упрощенная) архитектура модуля показана на рис. 1. Здесь имеется три процессора: УП – управляющий процессор, ИП – исполнительный (или вычислительный) процессор и КП – коммутационный процессор. Они связаны между собой через общую память, хотя каждый процессор имеет и свою локальную память. Структура УП достаточно проста, а в его функции входит анализ состояния элементов программы, реализующих свойства 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, при этом ИП может быть реализован как на стандартном микропроцессоре, так и схемах гибкой логики с возможностью перепрограммирования структуры процессора при переходе к выполнению другого объекта, либо в смешанном варианте. КП выполняет функцию передачи объектов (программ и данных) между модулями по высокоскоростным каналам, которых имеет не менее четырех. 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 взаимодействие между ними осуществляется при помощи системы аппаратно поддерживаемых очередей, размещаемых в локальной памяти процессоров.

Конфигурация связей между процессорами в общем случае не имеет значения, так как 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.

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, которое может быть произвольным, а также равномерной загрузкой всех ВМ, которая позволяет выполнять задачи, которые обладают способностью к распараллеливанию, на максимально возможном числе процессоров. Последнее 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), 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). 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 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, рассматриваемом 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, ее 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 меняется в процессе выполнения программы.

Исторически язык ЯРД «вырос» из ранее разработанного языка РЯД [5] с весьма неудачным синтаксисом, сохранив его семантику. Некоторое краткое описание основных характеристик языка ЯРД представлено в [6].

Каждый узел программной сети (объект) состоит из *дескриптора* и *тела*. Упрощенный пример структуры объекта приведен на рис. 2. 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 объектов), «объекты-соседи» (эти связи 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 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 физическо-

го 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».

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, 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 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.

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 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 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, изменив-

ший 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 ближайшее 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.

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», 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 обработ-

ку 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, 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 некоторо-

го 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 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.

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 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. 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, 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, присоединяя 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 та-

кими 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. На рис. 3 показаны графические обозначения объектов разных классов. 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 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. 4 показан пример очень простой программы, которая выполняет умножение двух матриц произвольного размера. 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. 5 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 результаты из отдельных элементов.

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 выполнения ска-

лярного произведения строк и столбцов матрицы 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.

4. Список литературы

[1] Glushkov V.M., Ignatyev M.B., Myasnicov V.A. and Torgashev V.A. Recursive Machines and Computing Technology. - In: 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. Москва, Наука, 1982, сс. 172-187.

[3] Плюснин В.У., Пономарев В.М., Торгашев В.А. Распределенные вычисления и машины с динамической архитектурой. В сб.: Актуальные проблемы развития архитектуры и программного обеспечения ЭВМ и вычислительных систем. Новосибирск, Вычислительный центр, 1983.

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

[5] РЯД – язык программирования для распределенных вычислений. Препринт №28. Академия наук СССР, Ленинградский научно-исследовательский вычислительный центр, Ленинград, 1984, сс. 3-48.

[6] Torgashev V.A. and Tsaryov I.V. A Parallel Programming Language for Dynamic Architecture Computers. - In Proc. of The Int. Conf. Parallel Computing Technologies. ReSCo, Moscow, 1993, p. 31-34.

Сведения об авторах.

1. Торгашёв Валерий Антонович, окончил Ленинградский электротехнический институт им. В.И.Ульянова (Ленина) – ЛЭТИ в 1961 г., доктор технических наук, профессор, заведующий лабораторией распределенных вычислительных структур Санкт-Петербургского института информатики и автоматизации РАН (СПИИРАН). Научные интересы: мультипроцессорные компьютеры с динамической архитектурой, суперкомпьютеры с асинхронным распределенным управлением, распределенные параллельные вычисления, процессоры с перепрограммируемой архитектурой, обработка сигналов в реальном масштабе времени
2. Царёв Игорь Владимирович, окончил Ленинградский электротехнический институт им. В.И.Ульянова (Ленина) – ЛЭТИ в 1970 г., старший научный сотрудник лаборатории распределенных вычислительных структур Санкт-Петербургского института информатики и автоматизации РАН (СПИИРАН). Научные интересы: Разработка языков программирования и компиляторов, параллельное программирование, распределенные вычисления, управление параллельными вычислительными процессами.

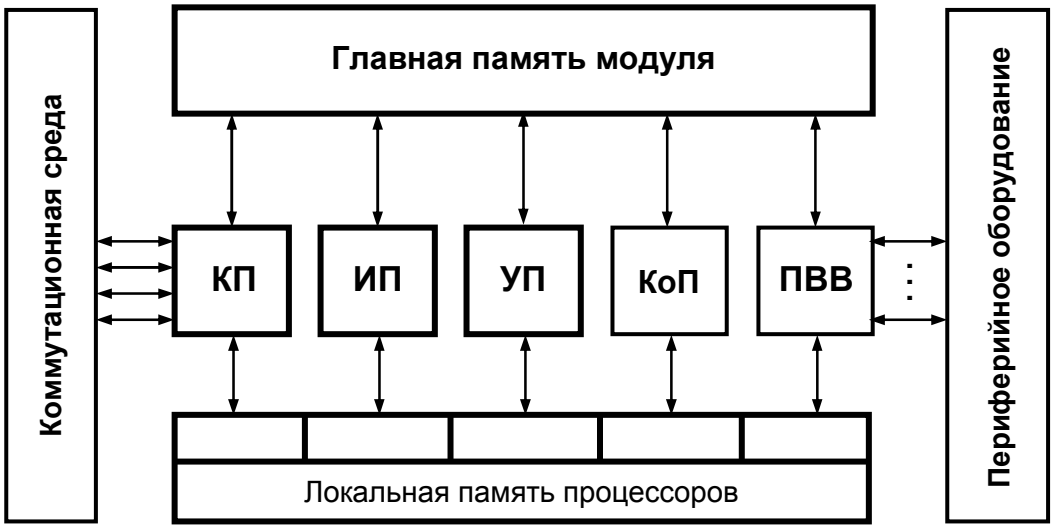

Рис. 1. Архитектура вычислительного модуля МДА

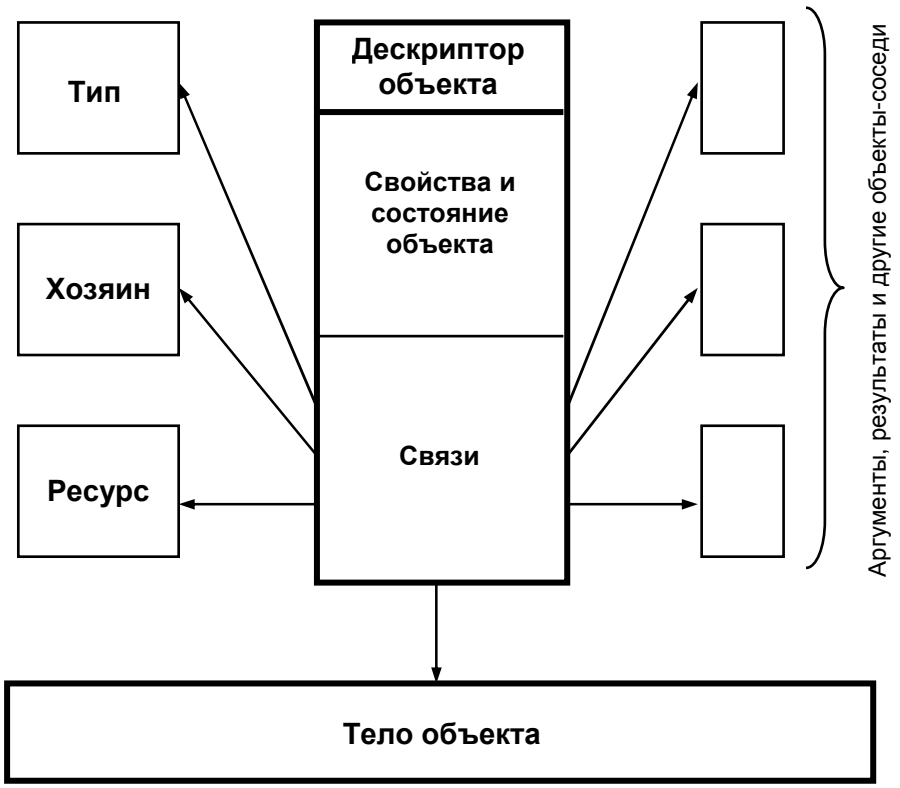

Рис. 2. Упрощенная структура объекта программной сети и его связей

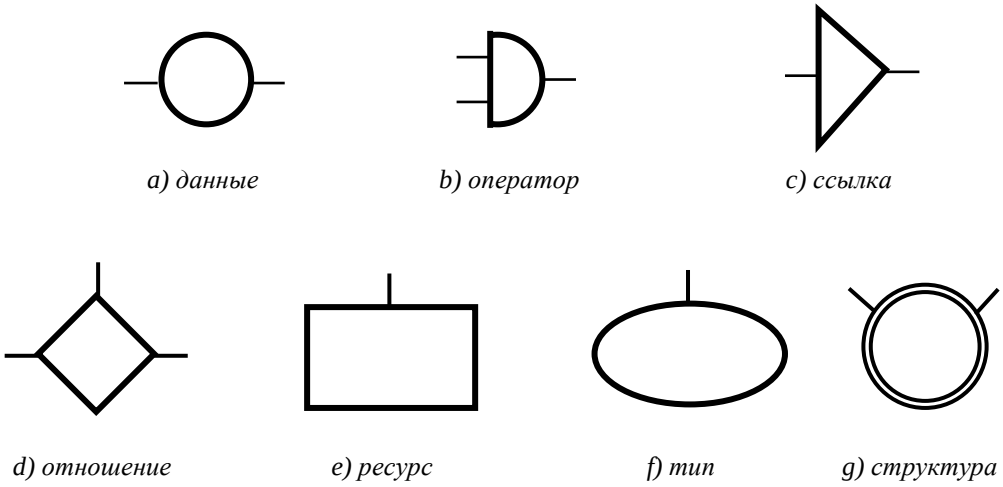

Рис. 3. Графические обозначения объектов различных классов
в языке ЯРД.

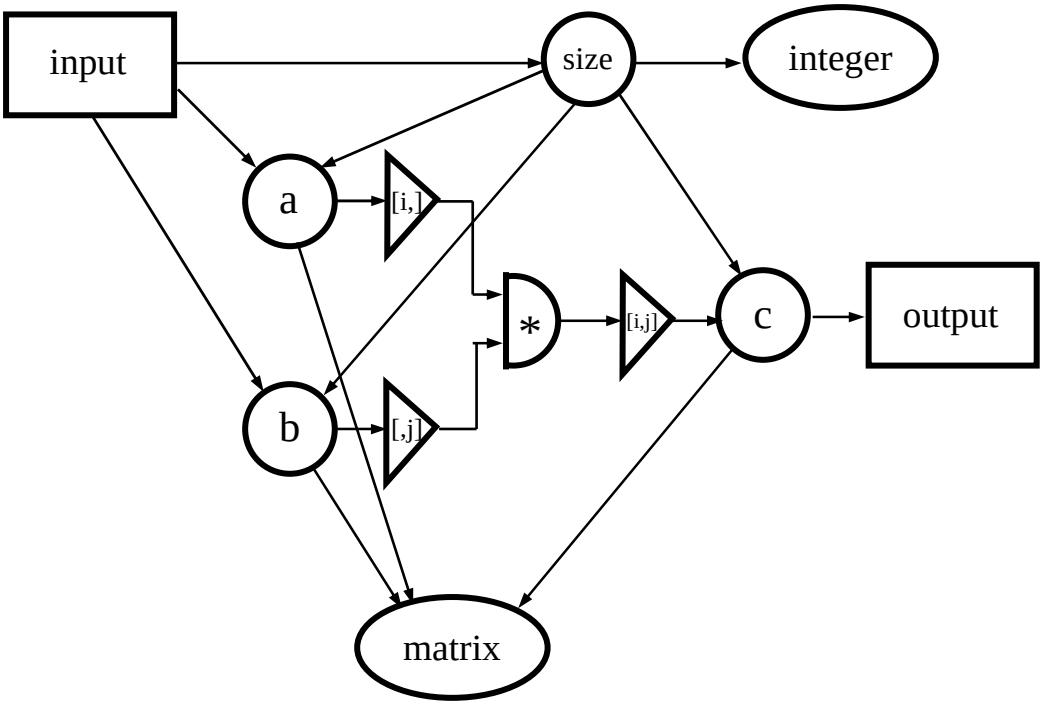

Рис. 4. Пример графического представления программы на языке ЯРД.

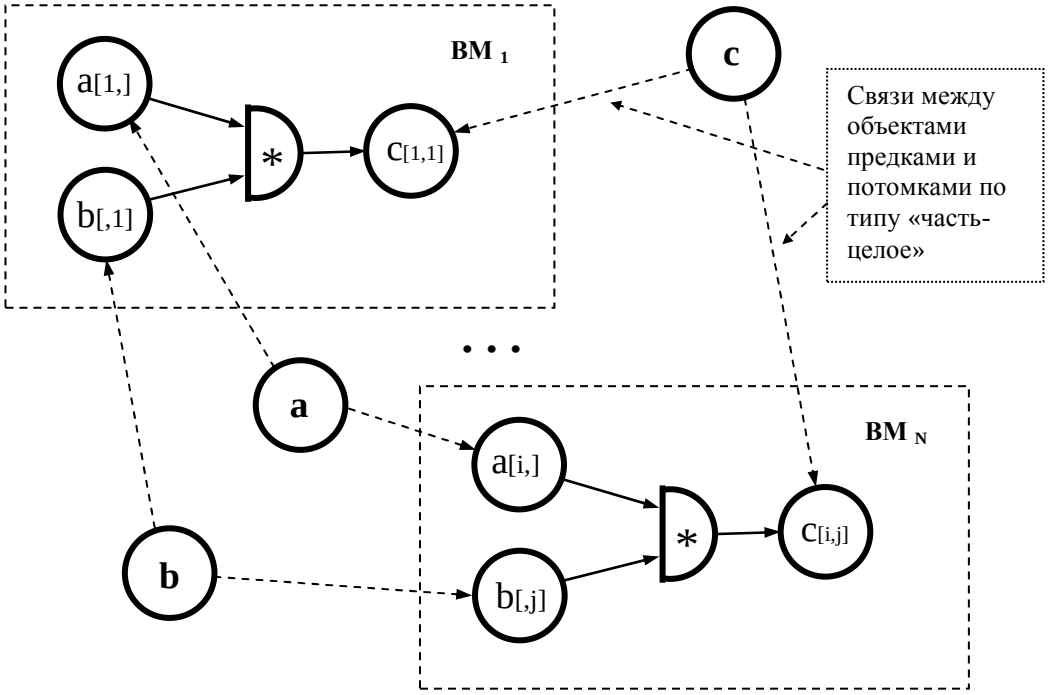

Рис. 5. Пример порождения групп объектов при автоматическом распараллеливании фрагмента программы.