МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)» (СГАУ)

ИСПОЛЬЗОВАНИЕ УЧЕБНО-ИССЛЕДОВАТЕЛЬСКОГО ПРОГРАММНОГО КОМПЛЕКСА АВТОМАТИЗАЦИИ МОДЕЛИРОВАНИЯ АЛГОРИТМОВ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ PGRAPH

Электронные методические указания
к лабораторным работам и самостоятельной работе

Работа выполнена по мероприятию блока 2 «Развитие и повышение
эффективности научно-инновационной деятельности» и
блока 3 «Развитие информационной научно-образовательной среды и инфраструктуры»
Программы развития СГАУ на 2009 – 2018 годы
по проекту «Разработка учебно-исследовательского комплекса автоматизации
моделирования алгоритмов параллельных вычислений»
Соглашение № 2/8 от 03 июня 2013 г.

САМАРА

2013

УДК 621.9.02
ББК 32.9я7
И 883

Составители: **Жидченко Виктор Викторович,**
Коварцев Александр Николаевич

Рецензент: д-р техн. наук, проф. В.С. Кузьмичев

Использование учебно-исследовательского программного комплекса автоматизации моделирования алгоритмов параллельных вычислений PGRAPH [Электронный ресурс]: электрон. метод. указания к лаб. и самост. работам / М-во образования и науки РФ, Самар. гос. аэрокосм. ун-т им. С. П. Королева (нац. исслед. ун-т); - авт.-сост. *В.В. Жидченко, А.Н. Коварцев.* - Электрон. текстовые и граф. дан. (4,7 Мбайт). - Самара, 2013. - 1 эл. опт. диск (CD-ROM).

В методических указаниях к лабораторным и самостоятельной работам студента приводится описание технологии визуального моделирования параллельных алгоритмов в системе PGRAPH, реализующей концепции технологии графосимволического программирования.

Методические указания ориентированы на студентов шестого факультета, обучающихся по магистерской программе направления 010300.68 «Фундаментальная информатика и информационные технологии» (ФГОС-3). Работа может быть полезна студентам других факультетов, аспирантам и инженерам, использующим параллельные вычисления для выполнения расчетов в научных исследованиях, а также при изучении следующих дисциплин: «Методы и средства визуального параллельного программирования», «Автоматизация тестирования» в 4 семестре.
Разработано на кафедре программных систем.

СОДЕРЖАНИЕ

Введение

Методы разработки параллельных алгоритмов коренным образом отличаются от традиционных методов создания последовательных кодов программ. Учитывая широкую распространенность вычислительных систем с распределенной памятью, для организации информационного взаимодействия между процессорами часто используется интерфейс передачи сообщений MPI. Необходимость равнозначного владения одновременно технологиями программирования на языках высокого уровня (C++, Fortran) и средством организации параллельных вычислений MPI отдаляет «конечных» пользователей от возможности участия в разработке параллельных программных приложений. Существует множество графических моделей описания алгоритмов параллельных вычислений. В области представления вычислительных алгоритмов наиболее удобной является форма, близкая описанию блок-схемами, которая реализована в технологии графосимволического программирования (ГСП). Технология ГСП, созданная в рамках научного направления автоматизации визуального программирования, развиваемого на кафедре программных систем СГАУ, является одним из способов наглядного представления алгоритмов программ в виде графов управления. Для разработки программ в технологии ГСП создана и развивается система визуального программирования PGRAPH, описание которой приводится в данных методических указаниях.

В указаниях представлены: краткое описание области визуальной разработки параллельных алгоритмов и программ, а также возможности и средства системы PGRAPH для решения задач в этой области.

1. Назначение средства PGRAPH моделирования алгоритмов параллельных вычислений.

Система PGRAPH – автоматизированная система проектирования и разработки программ с использованием технологии графосимволического программирования. Технология использует графическую форму представления алгоритмов, семантически близкую к блок-схемам, дополненную механизмами описания параллелизма и синхронизации. Пример представления алгоритма в технологии ГСП приведен на рисунке 1.1.

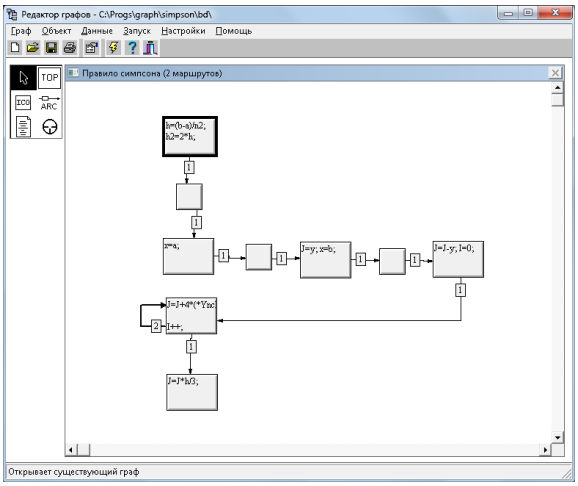


Рис. 1.1. Пример визуального представления алгоритма в системе ГСП PGRAPH

Алгоритм представляется в виде графа, называемого в технологии ГСП **агрегатом**. Вершинами графа служат подпрограммы на языке С или другие графы. Направленные дуги графа определяют передачу управления между вершинами. По графическому описанию алгоритма специализированный компилятор технологии ГСП строит исходный текст программы на языке С. Исполняемый файл формируется из этого исходного текста компилятором языка С++ с использованием библиотеки MPI. Система PGRAPH предоставляет визуальную среду для создания графического образа агрегатов и спецификации других объектов технологии ГСП, средства автоматизированного анализа проектируемых программ с целью повышения их надежности, а также средства автоматического синтеза исходных текстов

программ на основе объектов технологии ГСП. С помощью внешнего компилятора для языка программирования, на котором написаны базовые модули, система позволяет компилировать и запускать на выполнение созданные в ней программы. В текущей версии системы поддерживается язык программирования C++, наиболее широко используемый при создании программ для параллельных вычислений. Технология ГСП не накладывает ограничений на использование конкретного языка программирования, поэтому при необходимости в системе PGRAPH можно создавать программы на любом процедурном языке. Для этого необходимо подключить к системе внешний компилятор этого языка программирования.

1.1. Особенности разработки алгоритмов параллельных вычислений. Критические данные.

Специфической задачей при работе с параллельными вычислениями, коренным образом отличающей их от последовательных, является обеспечение согласованной работы параллельных процессов (синхронизация). Согласованность работы заключается, во-первых, в корректном использовании множества данных, с которыми одновременно работают несколько фрагментов параллельного алгоритма (параллельных ветвей), а во-вторых, в беспрепятственном выполнении параллельных ветвей при наличии их взаимного влияния друг на друга.

В последовательном алгоритме в каждый момент времени над любой переменной может выполняться только одна операция, определенная алгоритмом. При параллельных вычислениях одновременно выполняются несколько ветвей алгоритма, поэтому возможна ситуация, когда к одной и той же переменной в некоторый момент времени обращаются две или более ветвей. При этом одна ветвь может считывать значение переменной, а другая – изменять его. Такие переменные формируют множество **критических данных** параллельного алгоритма. Если моменты чтения и изменения

значения переменной не согласованы между собой, то результат операции чтения непредсказуем.

Для исключения подобных ситуаций разработаны различные механизмы синхронизации параллельных процессов: критические секции, семафоры, мониторы, сообщения /23/.

1.2. Синхронизация параллельных вычислений

Семафоры. Главная причина сложности решения проблемы взаимного исключения состоит в том, что проверки значений переменных, используемых для синхронизации, и изменения этих значений - независимые события. В то время как один процесс проверяет условия, разрешающие или не разрешающие войти ему в критический интервал, он не знает, начал ли другой процесс изменять значения переменных, вовлеченных в формирование условия, то есть, не изменилась ли истинность условия.

Решение проблемы взаимного исключения существенно упрощается, если ввести специальные программные средства для синхронизации, обладающие некоторыми качественно новыми свойствами. Дейкстра /24/ предложил механизм, включающий специальные переменные нового типа — семафоры и две операции P и V, аргументами которых могут быть переменные только типа семафоров. Область значений семафора — целые неотрицательные числа. Если область значений сужена до 0 и 1, семафор называют бинарным.

Операция V изменяет значение s семафора на значение s+1. Действие операции P определяется следующим образом:

если s □ 0, то P уменьшает значение s на единицу;

если s = 0, то P не изменяет значения s и не завершается до тех пор, пока некоторый другой процесс не изменит значение s с помощью операции V.

Существенно, что операции P и V считаются «неделимыми». По отношению к V это означает следующее. Операция V состоит из трех фаз:

1) считывание значения семафора из памяти;

2) увеличение значения семафора;

3) помещение нового значения в память.

Неделимость V(s) заключается в том, что с самого начала выполнения этой операции до ее завершения доступ к семафору s запрещен для всех других операций. Аналогично дело обстоит с операцией P. Обычно P предшествует критическому интервалу процесса, а V завершает его. В каждый момент времени, когда значение семафора s изменяется с 0 на 1, только одна из операций может завершиться и разрешить вход в критический интервал только одному процессу.

Механизм семафоров обеспечивает организацию любых схем взаимодействия процессов и к настоящему времени широко используется в операционных системах, встроен в ряд универсальных языков программирования. Тем не менее семафоры не очень удобны, так как они слишком «примитивны»: бесконтрольное использование семафоров запутывает структуру управления параллельного алгоритма; велика вероятность ошибки при описании сложных взаимодействий.

Критические секции (интервалы). Механизм критических секций привязан к тем общим ресурсам, через которые или из-за которых необходима организация взаимодействия параллельных процессов.

В этом подходе общие ресурсы представлены переменными, которые имеют специальный описатель shared (общие). В контексте языка Паскаль описания таких переменных имеют вид

var v : shared T,

где v — идентификатор переменной; T — тип переменной.

Процесс может использовать общие переменные только внутри оператора, называемого условным критическим интервалом. Он имеет вид

region v when B do S,

где B — условное выражение; S — оператор.

Процесс входит в условный критический интервал (только один процесс может это сделать) и вычисляет значение B. Если оно — истина, то процесс

начинает исполнение оператора. B противном случае процесс уходит из критического интервала и задерживается до тех пор, пока другой процесс не завершит своих вычислений в критическом интервале. B этом случае задержанный процесс может вновь войти в критический интервал и вычислить B. Этот цикл продолжается, пока B не станет истинным, после чего будет исполнен оператор S. B конкретной реализации задержанные операторы образуют очередь, из которой они выбираются в соответствии с некоторой дисциплиной, принятой в этой реализации. (B реализации семафоров поступают аналогично.)

B качестве симметричной конструкции используется оператор

region v do S await B.

Этот оператор задерживает процесс после исполнения S до тех пор, пока B не станет истинным. Если B тождественно истинно, то оба условных критических интервала имеют вид region v do S.

Мониторы. Мониторы – это следующий шаг в повышении уровня синхропримитивов, привязке их к данным и централизации управления параллельными процессами. Монитор представляет собой совокупность управляющих переменных, связанных с некоторыми общими ресурсами, и процедур запросов и освобождения этих ресурсов. Монитор «отторгает» от процессов все их критические интервалы, связанные с данными ресурсами, и превращает их в свои процедуры. Процедуры монитора вызываются указанием имени процедуры и имени монитора. Они доступны всем процессам в том смысле, что любой из процессов может попытаться вызвать любую процедуру, но только один из процессов может войти в процедуру, остальные должны дожидаться того момента, когда будет завершен предыдущий вызов. Процедуры монитора могут содержать только переменные, локализованные в теле монитора.

Эти ограничения предостерегают от ошибок, возникающих при использовании семафоров. Для того, чтобы иметь возможность задержать вызывавший данную процедуру процесс до того момента, когда освободится

занятый ресурс, и для того, чтобы сообщать об освобождения ресурсов, используются управляющие переменные нового типа «условие» (condition). Эти переменные используются внутри процедур монитора и только в сочетании с операциями wait и signal, которые следуют за идентификатором переменной и отделены от нее точкой: cond.wait, cond.signal. Переменная типа «условие» объявляется для каждой причины, могущей вызвать задержку. Она не имеет хранимых в памяти значений, а представляет собой очередь процессов, ждущих выполнения данного условия.

Синхронизация сообщениями. В этом методе синхронизации используются специальные средства для передачи сообщений от процессов к процессам. Имеются средства маршрутизации потоков сообщений и их упорядочения после прибытия к месту назначения, если они не могут быть восприняты немедленно. Процессы снабжаются специальными операциями для формирования и отсылки сообщений, операциями ожидания сообщений и их обработки. Сообщения могут классифицироваться по приоритетам и другим признакам.

Примером может служить механизм, получивший название почтовый ящик. Если процессу А необходимо взаимодействовать с процессом В, процесс А просит исполняющую систему образовать почтовый ящик для передачи сообщения. После этого процесс А помещает сообщение в почтовый ящик, откуда процесс В может взять его в любое время. С программной точки зрения почтовый ящик - это структура данных, для которой фиксированы правила работы с ней. Она включает заголовок, содержащий описание почтового ящика, и несколько гнезд, в которые помещаются сообщения. Число и размеры гнезд задаются при образовании почтового ящика. Правила работы с ящиком различаются в зависимости от его сложности. В простейшем случае сообщения передаются в одном направлении. Процесс А заполняет свободные гнезда, а если все гнезда заполнены, то процесс обычно ждет их освобождения. Процесс В может получать сообщение до тех пор, пока в ящике имеются заполненные гнезда.

Если есть необходимость получать от В подтверждения о приемке сообщений, то можно организовать почтовый ящик так, чтобы он обеспечивал двустороннюю связь. В этом случае гнездо ящика содержит либо сообщение от А, либо ответ от В. Может случиться, что процесс А формирует сообщение с большей скоростью, а процесс В не может ответить на них, так как все гнезда уже заняты сообщениями от А. Такой ситуации можно избежать, если потребовать, чтобы объекты пересылались через те же гнезда, в которых находились вызвавшие их сообщения. Для организации связи многих процессов с одним служат многовходовые почтовые ящики, через которые несколько процессов посылают сообщения, и устанавливается некоторый порядок их приема.

При непосредственной связи процессов через почтовые ящики процессы должны знать имена «своих» ящиков. Это неудобно в тех случаях, когда к некоторому процессу обращаются многие, безразлично какие процессы (например, обращение процессов к процедурам). Чтобы освободиться от имени ящиков, вводят порты, которые служат связующим звеном между процессами и ящиками.

Процессы имеют порты двух типов — входные и выходные. Каждый почтовый ящик (двусторонний) связан с одним входным и одним выходным портом. Каждое гнездо ящика находится в данный момент времени в одном из четырех состояний: ожидает сообщения, содержит сообщение, ожидает ответа, содержит ответ. По команде «образовать почтовый ящик», содержащейся в некотором процессе, создается ящик. С помощью команды «связать» почтовый ящик связывается с некоторым портом. Для того, чтобы послать сообщение через определенный порт, применяется команда «послать», которая проверяет, есть ли в почтовом ящике, соединенном с указанным портом, гнездо в состоянии ожидания сообщения (или ответа, если посылается ответ). Если гнездо имеется, то сообщение записывается в гнездо, и последнее меняет свое состояние. Для запроса сообщения используется команда «получить» с параметрами, указывающими имена портов. Эта

команда просматривает почтовые ящики указанных портов, разыскивая гнездо, содержащее сообщение (или ответ). Найденное сообщение переписывается в память, отведенную для запросившего его процесса. Почтовый ящик может быть уничтожен специальной командой.

Почтовые ящики более удобны для обмена данными между процессами, чем для синхронизации критических участков. Однако их можно использовать и для второй цели, для чего необходимо организовать процесс, который распределяет разрешения на вход в критические участки, и устроить общение с ним для других процессов через почтовые ящики. Такой процесс будет напоминать монитор.

Перечисленные средства имеют низкий уровень и не всегда удобны для пользователя. Прямо они обычно не используются, а «поддерживают» более компактные языковые конструкции.

Существующие механизмы синхронизации обеспечивают корректную обработку совместно используемых данных несколькими процессами, однако задача выделения из множества данных тех из них, доступ к которым необходимо синхронизировать, как правило, возлагается на человека. Предполагается, что разработчик параллельного алгоритма заранее знает, исходя из логики алгоритма, какие данные потребуют синхронизации. На практике это не всегда так. Алгоритмы, требующие распараллеливания, обычно имеют сложную структуру и оперируют большим числом переменных. Разработчику параллельного алгоритма оказывается сложно отследить все информационные зависимости между вычислительными модулями. Ситуация еще больше усложняется при коллективной разработке параллельного алгоритма. Более того, возможны ошибки при использовании механизмов синхронизации, например, применение операции P к одному семафору и операции V – к другому. Ошибки синхронизации могут возникать не при каждом запуске программы и являются трудно обнаруживаемыми. В связи с этим, актуальна задача автоматизированного поиска критических данных в параллельных алгоритмах.

Асинхронные вычисления. Рассмотренные выше методы организации параллельных вычислений можно охарактеризовать как параллельно-последовательные: их основу составляют последовательность параллельных операторов или параллельно исполняемые последовательные ветви (процессы). Если из параллельно-последовательного процесса удалить дополнительные средства распараллеливания, то он становится обычным последовательным. Таким образом, можно считать отправной точкой этих методов последовательно-алгоритмическую организацию вычислений.

Противоположный подход состоит в том, что все действия (операторы) изначально считаются независимыми, параллельно исполняемыми, а формирование алгоритма организуется с помощью ограничений, налагаемых на порядок исполнения действий.

Метод организации вычислений, в котором на независимые операторы налагаются ограничения на порядок их выполнения в форме явно указываемых или неявно подразумеваемых индивидуальных условий готовности, связанных с каждым оператором, получил название асинхронных вычислений. Условия готовности динамически проверяются и разрешают или не разрешают (но не предписывают) начать выполнение операторов, с которыми данные условия связаны. Условия готовности берут на себя всю организацию управления, так что в асинхронных вычислениях можно не разделять средств управления на средства, формирующие последовательности операторов и параллельные ветви, и на средства синхронизации. Из сказанного следует, что асинхронный принцип организации вычислений является в некотором смысле дополнительным по отношению к параллельно-последовательному, и оба принципа эквивалентны в том смысле, что могут моделировать друг друга. Относительные достоинства и недостатки этих методов вычислений зависят от специфики решаемых задач и от свойств ЭВМ, на которых будут исполняться вычислительные программы.

Условия готовности операторов в асинхронных вычислениях могут формулироваться в разных терминах. В зависимости от этого можно выделить разные типы асинхронного управления.

Событийное управление основано на том, что условия готовности формулируются как логические функции от некоторых событий. Событием может быть инициирование или завершение какого-либо оператора программы (программные события), а также прерывание или сигнал об освобождении некоторого ресурса (системные события).

При потоковом управлении действие (оператор или операция) может выполниться, если готовы все необходимые для него аргументы (операнды). Условие готовности в этом случае носит стандартный характер и не выписывается явно в программе, а неявно подразумевается. В обратном потоковом управлении действие может выполниться, если его результат необходим в качестве аргумента для некоторого другого действия. В этом случае второе действие как бы вызывает первое в качестве процедуры.

В динамическом управлении условие готовности является логическим выражением, зависящим от переменных той же памяти, над которой определены операторы программы.

Все три типа асинхронного управления «универсальны» в том смысле, что позволяют запрограммировать любую вычислимую функцию. Все три типа управления позволяют достичь одинаковой степени параллелизма в программах, но на практике каждый из них оказывается приемлемым на определенном уровне сложности распараллеленных фрагментов (операции, операторы, модули). Возможно их сочетание в разных комбинациях.

Асинхронные вычисления удобны при моделировании реактивных систем (систем, взаимодействующих с внешней средой посредством обмена сообщениями) и систем реального времени.

В технологии ГСП используется синхронизация сообщениями совместно с некоторыми механизмами мониторной синхронизации. Визуально синхронизация описывается с помощью направленных дуг, соединяющих на

графе алгоритма две вершины, между которыми пересылается сообщение. Направление дуги совпадает с направлением передачи сообщения. В одну вершину могут входить дуги от нескольких других вершин. В этом случае для вершины задается семафорный предикат – логическая функция, которая определяет, при какой комбинации входящих сообщений вершина должна запуститься на выполнение.

1.3. Взаимные блокировки в параллельных вычислительных процессах. Тупики, дедлоки, клинчи.

Введение синхронизации в параллельный алгоритм приводит к возникновению другого типа ошибок – тупиков (взаимных блокировок, дедлоков, клинчей).

Синхронизация подразумевает наличие механизмов блокировки доступа к общим ресурсам (данным, устройствам ввода-вывода и т.п.) и ожидания разрешения такого доступа. Тупики возникают, когда два или более параллельных процессов ожидают освобождения ресурсов, которые никогда не будут освобождены. Например, процесс А блокирует доступ к ресурсу А1 и ожидает освобождения ресурса В1, а выполняющийся одновременно с ним процесс В блокирует ресурс В1 и ожидает освобождения ресурса А1. В результате процессы А и В приостанавливают работу и не могут возобновить ее вновь, т.е. взаимно блокируют друг друга.

Известны методы поиска тупиков в абстрактных моделях параллельных алгоритмов типа «задача-канал» и сетях Петри, однако на практике, как правило, борьба с тупиками сводится к разработке средств обнаружения тупиков уже в процессе вычислений и устранения их последствий. Такой подход эффективен для обеспечения надежности параллельных программ, например, в системах управления реального времени, когда не так важны причины возникновения тупика, как важно не допустить полной остановки программы. В программах вычислительного характера, итогом работы которых является некоторый численный результат, возникновение тупика не

позволяет получить этот результат, а значит, свидетельствует о наличии ошибки в алгоритме. Вычисления, требующие распараллеливания, в силу их сложности, выполняются относительно долго. Безрезультатная остановка вычислений из-за тупика, с последующим трудоемким выяснением причин этого тупика, приводит к непроизводительным затратам времени. Поэтому важно обнаруживать тупики до запуска вычислений.

В системе графосимволического программирования PGRAPH реализован метод автоматического поиска тупиков на основе анализа графа алгоритма с учетом дуг синхронизации, введенных разработчиком алгоритма для устранения критических данных.

1.4. Стандартные средства описания параллельных процессов

Несмотря на существование множества различных технологий и средств параллельного программирования, на момент написания настоящего пособия в области параллельных вычислений доминируют две технологии: MPI и OpenMP [3].

MPI используется в основном при разработке параллельных программ для систем с распределенной памятью (массивно-параллельных систем, кластеров и GRID-систем), в то время как OpenMP – для систем с общей памятью (SMP-систем).

Существуют реализации указанных технологий для различных языков программирования и различных аппаратных платформ, что позволяет при написании программы максимально абстрагироваться от особенностей вычислительной системы, на которой она будет исполняться.

Технология OpenMP. В технологии OpenMP вычисления распределяются между несколькими потоками, имеющими равноценный доступ к общей области памяти. Для передачи данных между потоками не требуется специальных конструкций. Достаточно использовать для чтения и записи данных различными потоками область памяти с одним и тем же адресом.

Технология OpenMP формирует так называемый «пульсирующий» параллелизм. Выполнение программы начинается в одном главном потоке, называемом «мастер-поток». В некоторой точке мастер-потока может быть порожден набор других потоков, при этом вычисления, описываемые программой, распределяются между порожденными потоками и мастер-потоком и выполняются параллельно. Затем порожденные потоки завершаются и вычисления продолжаются в мастер-потоке (рисунок 1.2).

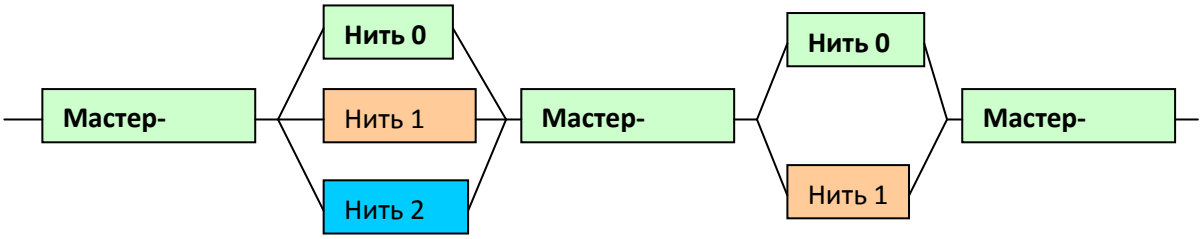


Рисунок 1.2. Модель «пульсирующего» параллелизма технологии OpenMP

Каждый поток также может породить набор потоков, после завершения которых вычисления продолжатся в данном потоке.

Для описания параллельных фрагментов программы технология OpenMP использует директивы компилятора вида:

#pragma omp <имя_директивы> [clause,…]

Таким образом, для успешного применения технологии необходимо, чтобы компилятор ее поддерживал. Существует множество компиляторов различных производителей, а также компиляторов с открытым исходным кодом (Open Source), поддерживающих данную технологию. Если технология не поддерживается компилятором, то ее директивы будут игнорироваться. Это позволяет в некоторых случаях создавать универсальный исходный код программы, компилируемый как параллельный или последовательный в зависимости от имеющегося компилятора.

При наличии соответствующего компилятора программа может быть написана на любом языке программирования. На практике для написания программ вычислительного характера используются языки C и Fortran.

Выделяют следующие преимущества технологии OpenMP:

- возможность поэтапного распараллеливания программы путем вставки директив OpenMP в ее различные участки без изменения структуры и основного исходного кода программы;

- возможность создавать программы с универсальным исходным кодом, компилируемые как последовательные или параллельные в зависимости от используемого компилятора;

- возможность использования единого способа распараллеливания вычислений в программах на различных языках программирования, если компилятор соответствующего языка поддерживает технологию OpenMP;

- возможность написания параллельных программ без учета особенностей вычислительной системы – оптимальная реализация директив OpenMP на конкретной платформе выполняется в этом случае разработчиками платформы и обеспечивается предоставляемым ими компилятором.

Технология MPI (Message Passing Interface) применяется для создания программ, работающих в системах с распределенной памятью. Каждый процесс параллельной MPI-программы имеет собственную область памяти. Обмен данными между процессами выполняется явными операциями пересылки данных в виде сообщений. Технология MPI определяет перечень, семантику и синтаксис функций пересылки сообщений. Спецификация функций не привязана к конкретным языкам программирования, поэтому обеспечивается переносимость программ.

Фактически реализация MPI для конкретной системы – это библиотека функций на некотором языке программирования и набор программ, обеспечивающих среду выполнения MPI-приложений. Наиболее распространены реализации библиотек MPI для языков C и Fortran. Как правило, производители вычислительных систем предоставляют библиотеки MPI, оптимизированные для выполнения на конкретной вычислительной системе. Существуют и свободно распространяемые версии библиотек MPI.

Среда выполнения MPI-приложений обеспечивается программами-«демонами», работающими на всех узлах вычислительной системы и осуществляющими запуск вычислительных процессов на конкретных узлах. Кроме того, в состав среды выполнения часто входит набор утилит для ее конфигурирования и мониторинга.

MPI-приложение состоит из одного исполняемого файла, который копируется и запускается на заданном числе узлов вычислительной системы. Таким образом, на различных узлах запускается одна и та же программа. Среда выполнения присваивает каждому экземпляру программы уникальный номер, называемый рангом процесса. В составе библиотеки MPI имеется функция MPI_Comm_rank для определения ранга процесса, в котором вызвана эта функция. Путем вызова функции MPI_Comm_rank отдельный процесс может определить свой ранг и в зависимости от его значения выполнить уникальную последовательность действий. Типичная MPI-программа содержит набор операторов условного перехода, обеспечивающих выполнение различных фрагментов программы (параллельных ветвей) для различных значений ранга текущего процесса.

1.5. Визуализация представления алгоритмов параллельных вычислений

Основой подавляющего большинства графических способов представления параллельных процессов является форма представления в виде графа, то есть совокупности вершин (узлов), соединенных между собой дугами (ребрами). В отличие от текстовой формы записи, в которой объекты (символы и слова) образуют последовательность, а каждый объект связан только с левым и правым «соседом», графовая форма позволяет наглядно изображать более сложные взаимосвязи, поскольку в ней каждый объект может соединяться с несколькими другими объектами. В этом смысле текстовая форма одномерна, в то время как графовая форма – многомерна. Возможность варьировать геометрические размеры, форму и цвет вершин,

внешний вид и толщину дуг, изменять взаимное расположение вершин без изменения топологии графа значительно увеличивают выразительные возможности графовой формы представления.

Графические модели обычно являются ориентированными графами, в которых дуги определяют направление передачи данных или зависимость между вершинами. Вершины и дуги обычно снабжаются текстовыми аннотациями, которые именуют их, перечисляют их содержимое или свойства.

Различные графические модели отличаются друг от друга семантикой вершин и дуг. Основные классы графических моделей параллельных процессов приведены на рисунке 1.3.

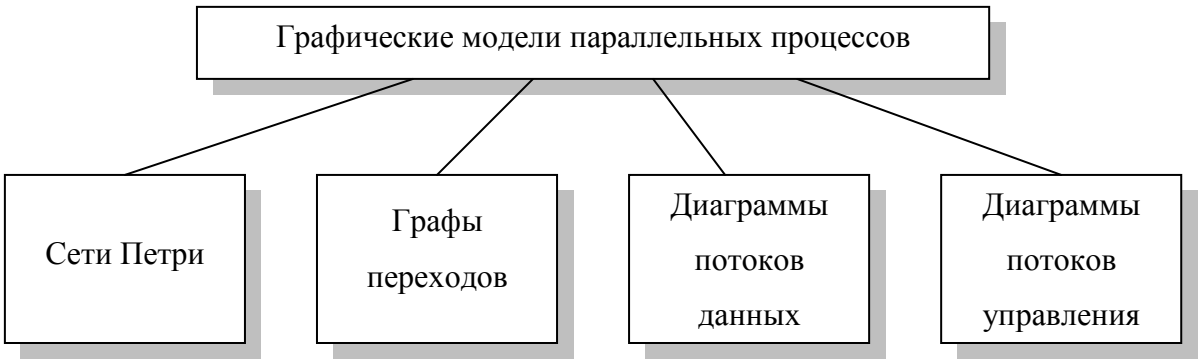


Рис. 1.3. Основные классы графических моделей параллельных вычислительных процессов

Существуют примеры систем (**сети Петри**), где программа на визуальном языке состоит из потока управления, потока данных и множества интерпретаций узлов (то есть процедур, запускаемых при достижении узла). Граф потока управления языка в этом случае обеспечивает представление последовательных потоков, переключение управления, параллелизм и синхронизацию. Параллелизм поддерживается в основном за счет описания того, как несколько маркеров, описывающих поток данных, могут распространяться по графу. На язык и систему программирования могут возлагаться задачи уже на этапе проектирования и «визуального кодирования» добиться правильности (в частности детерминированности) и эффективности параллельных программ.

Возможность формального математического описания сетей Петри и их функционирования в терминах теории множеств, а также достаточно сильное абстрагирование от деталей и специфики моделируемого процесса определили широкую популярность сетей Петри для описания различных дискретных динамических систем. Разработано множество подклассов и обобщений сетей Петри, адаптированных для моделирования определенных классов систем, подробно изучены их свойства.

Недостатком сетей Петри при описании вычислительных процессов является необходимость перехода к терминам модели, таким как переходы, места, разметка. Такой переход не всегда очевиден. Кроме того, наглядно изобразить функционирование сети Петри можно только в динамике срабатываний ее переходов. Статическое изображение сети обладает низкой наглядностью. Пример использования формализма сети Петри представлен на рисунке 1.4.

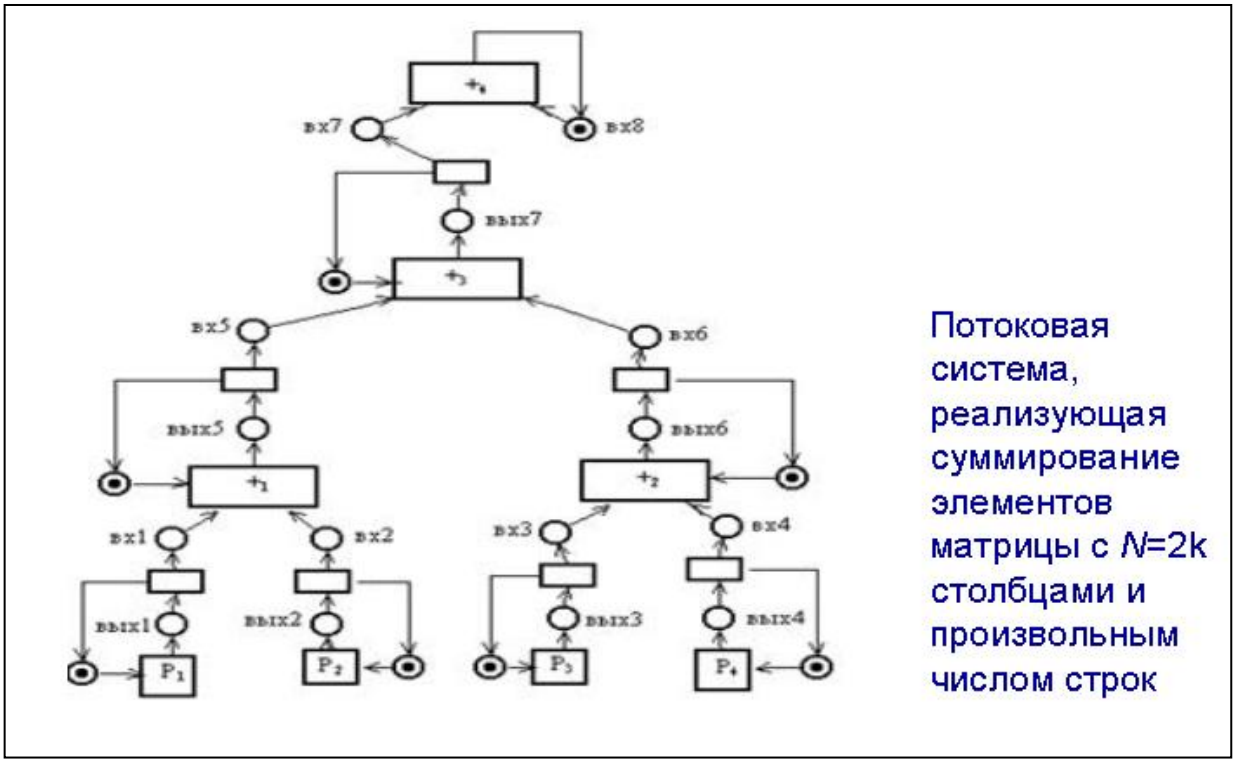


Рис. 1.4. Пример Сети Петри

Графы переходов (диаграммы состояний, диаграммы изменений состояний) представляют собой графическую форму описания конечных

автоматов. Вершины графа переходов соответствуют состояниям автомата, а дуги определяют возможные переходы из одного состояния в другое. Традиционно графы переходов используются для описания «последовательных» автоматов, так как одновременные переходы по двум и более дугам, исходящим из одной вершины, считаются запрещенными. Если такие переходы разрешены, то графы переходов, в которых допустимо одновременное существование нескольких «активных» вершин, называются графами переходов с параллелизмом. Графические модели на основе графов переходов используются в SWITCH-технологии [4] и графическом языке Statecharts.

В Институте кибернетики имени В.М.Глушкова была разработана Р-технология программирования, использующая графическую модель описания алгоритмов, близкую к автоматным, и получившую название Р-схем [12]. Р-схема – нагруженный по дугам ориентированный граф, изображаемый с помощью горизонтальных и вертикальных линий и состоящий из структур с одним входом и одним выходом. Построение Р-схем осуществляется из графических структур двух видов (рисунок 1.5, а) с помощью операций последовательного, параллельного и вложенного их соединения (примеры приведены на рисунках 1.5, б), в) и г) соответственно).

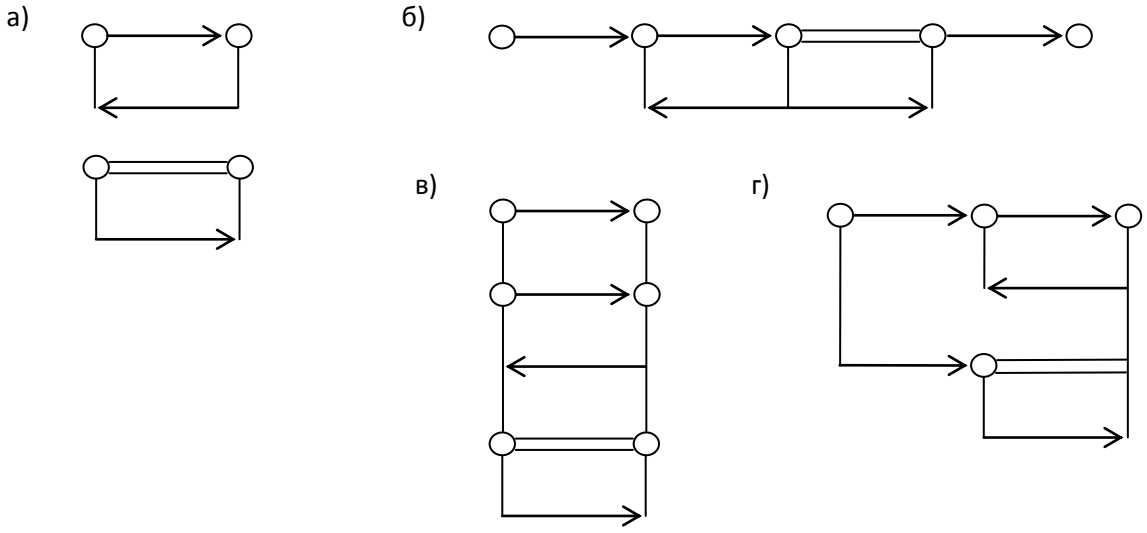


Рис. 1.5. Пример построения Р-схем

Разработан и введен в действие ГОСТ 19.005-85, определяющий внешнюю форму представления Р-схем.

В Р-схемах используется два принципиально различных алфавита для задания логики алгоритма (графический алфавит) и функциональных операторов, связанных с обработкой данных (текстовый алфавит). Задаваемый Р-схемой ориентированный граф заменяет собой операторы управления, представляемые в текстовых языках программирования инструкциями if, while, repeat, case и д.р. Направленная дуга изображает переход из одного состояния вычислительного процесса в другое. Над дугой записывается логическое условие прохождения по дуге, а запись под дугой относится к функциональной составляющей и определяет выполняемые при проходе по дуге действия по обработке данных. Текстовые записи осуществляются на одном из традиционных языков программирования, например С или Паскаль.

В ранних визуальных языках при проектировании программы на графовых схемах могла отображаться потенциальная возможность параллельного выполнения кода. Обычно в этих языках параллельные дуги графа описывали потенциально параллельные участки программы. При этом реальное создание эффективного параллельного кода возлагалось не на программиста, а на систему программирования. На следующем этапе развития визуальных языков параллельного программирования графы используются уже для создания настоящей параллельной программы с описанием взаимодействия ее процессов. Во многих случаях такие языки базируются на диаграммах потоков данных.

На диаграммах можно показать создание данных одним из последовательных вычислений для нужд других процессов. Параллельность, обычно, задается дополнительной параллельной дугой. При программировании вначале рисуется граф потоков данных решения данной задачи, а на следующем этапе за счет текстового описания портов ввода и вывода процесса аннотируются узлы графа (последовательные процессы), а также пишутся правила запуска этих узлов.

Диаграмма потоков данных позволяет наглядно изобразить интерфейс по данным между различными участками вычислительного процесса. Множество входящих дуг каждой вершины определяет ее входные данные. Результаты вычислений, производимых в вершине, передаются по исходящим дугам другим вершинам. Диаграммы потоков данных используются, например, в системах CODE 2.0, Paralex.

Диаграммы потоков управления описывают передачу управления между компонентами вычислительного процесса. Модель в этом случае также представляется в виде ориентированного графа, в котором дуги определяют порядок запуска на исполнение вершин графа. Каждая вершина соответствует некоторой последовательности вычислительных операций или другому графу, если модель допускает иерархию. Простейшим примером таких моделей являются блок-схемы. Достоинством диаграмм потоков управления является наглядность описания вычислительного процесса, а основным недостатком – непрозрачность интерфейса по данным между вершинами. Примерами моделей этого типа являются графический язык системы HeNCE [26], технология графосимволического программирования [18].

2. Средство визуального моделирования параллельных алгоритмов PGRAPH

2.1. Представление моделей алгоритмов в графической форме.

2.1.1. Концептуальная модель последовательного алгоритма

В технологии ГСП подразумевается, что любой алгоритм разрабатывается в рамках некоторой предметной области.

Под **предметной областью** программирования (ПО) будем понимать некоторую среду программирования, имеющую общую цель - разработку программного обеспечения автоматизации расчетов в некоторой области практических интересов (авиационные двигатели, бизнес, и т.д.) - и включающую общую область данных и общую область знаний.

Модель алгоритма в технологии ГСП представляется четверкой <D, F, P, G>, где D – множество данных некоторой предметной области, F – множество операторов, определенных над данными предметной области, P – множество предикатов, действующих над структурами данных предметной области, G = {A, Ψ, R} – ориентированный помеченный граф, называемый **агрегатом**. A = {A1, A2, …, An} – множество вершин графа. Каждая вершина Ai помечена локальным оператором fi(D) ∈ F. На графе задано множество дуг управления Ψ = {Ψ1i1, Ψ1i2, …, Ψjm}. R – отношение над множествами вершин и дуг, определяющее способ их связи. Дуга управления, соединяющая любые две вершины Ai и Aj, имеет три метки: предикат pij(D) ∈ P, приоритет k(Ψij) ∈ N и тип дуги T(Ψij) ∈ N.

Развитие процесса вычислений, описываемого моделью, ассоциируется с переходами из вершины в вершину по дугам управления. При этом переход по дуге управления возможен лишь в случае истинности предиката, которым она помечена. Если несколько предикатов, помечающих исходящие из вершины дуги, одновременно становятся истинными, переход осуществляется

по наиболее приоритетной дуге. Функционирование модели начинается с выполнения оператора f0(D), помечающего начальную вершину A0.

2.1.2. Концептуальная модель параллельного алгоритма.

Для расширения классической модели графосимволического программирования с целью описания в ней параллельных алгоритмов в модель вводятся дуги нескольких типов.

Тип дуги Ψij определяется как функция T(Ψij) ∈ {1,2,3}, значения которой имеют следующую семантику:

T(Ψij) = 1 – последовательная дуга (описывает передачу управления на последовательных участках алгоритма);

T(Ψij) = 2 – параллельная дуга (обозначает начало нового параллельного фрагмента алгоритма)

T(Ψij) = 3 – терминирующая дуга (завершает параллельный фрагмент алгоритма).

Графическое изображение дуг различных типов приведено на рисунке 2.1.

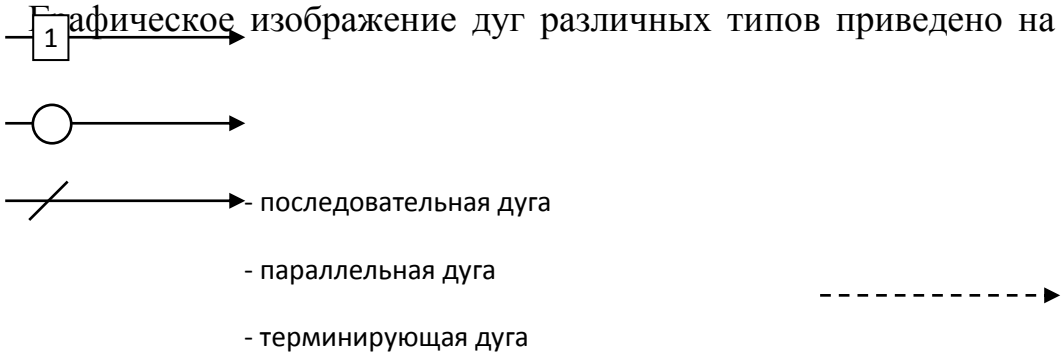


Рисунок 2.1 – Обозначение дуг различных типов в графической модели параллельных вычислений

Для описания параллелизма вводится понятие параллельной ветви β - подграфа графа G, начинающегося параллельной дугой (тип этой дуги T(Ψij) = 2) и заканчивающегося терминирующей дугой (тип этой дуги T(Ψij) = 3). β = <Aβ, Ψβ, Rβ> , где Aβ – множество вершин ветви, Ψβ – множество дуг управления ветви, Rβ – отношение над множествами вершин и дуг ветви, определяющее способ их связи.

Дуги, исходящие из вершин параллельной ветви β, принадлежат также ветви β. При кодировании алгоритма, описанного с помощью предлагаемой модели, каждая параллельная ветвь порождает отдельный процесс – совокупность подпрограмм, исполняемых последовательно на одном из процессоров параллельной вычислительной системы.

Графическая модель обычно содержит несколько параллельных ветвей, каждая из которых образует отдельный процесс. В этом смысле модель параллельных вычислений можно представить как объединение нескольких параллельных ветвей:

$$G = U\ \beta k, \quad (2.2)$$

где βk – параллельные ветви графа G.

Таким образом, распараллеливание вычислений возможно только на уровне граф-модели. Вычисления в пределах любого актора выполняются последовательно.

Число параллельных ветвей в модели фиксируется при ее построении, при этом максимальное количество ветвей не ограничивается. Каждая ветвь имеет ровно один вход и один выход, для обозначения которых в граф-модели используются два типа дуг: параллельная дуга и терминирующая дуга (рисунок 2.1, а).

Для описания правил построения граф-модели введем следующую систему обозначений:

To(Ai) – список дуг, входящих в вершину Ai

Fr(Ai) – список дуг, исходящих из вершины Ai

H(βj) – начальная вершина ветви βj

| L | – число элементов в списке L

Переход по параллельной дуге начинает работу параллельной ветви, переход по терминирующей дуге – заканчивает ее работу. Параллельная дуга не содержит предиката, т. е. переход по ней происходит безусловно.

$$\forall \Psi ij \in \Psi,\ T(\Psi ij) = 2 : Pij \equiv 1 \quad (2.4)$$

Из вершины может исходить не менее двух параллельных дуг. Если из вершины исходит параллельная дуга, то дуги других типов (обычная или терминирующая) не могут исходить из данной вершины.

$$T(\Psi ij0) = 2 \rightarrow \forall j \neq j0 : T(\Psi ij) = 2, \mid Fr(Ai) \mid \geq 2 \qquad (2.5)$$

Входящие в вершину дуги не могут быть одновременно параллельными и терминирующими:

$$T(\Psi i0j0) = 2 \rightarrow \forall i : T(\Psi ij0) \neq 3 \qquad (2.6)$$

Функционирование модели начинается с запуска единственной ветви, называемой мастер-ветвью (мастер-процессом). Обозначим мастер ветвь $\beta 0$. В вершинах мастер-ветви, имеющих исходящие параллельные дуги, порождаются новые параллельные ветви. Вершины этих ветвей также могут иметь параллельные дуги, таким образом, допускается вложенность параллельных ветвей. Ветви, породившиеся в одной вершине некоторой ветви, должны терминироваться также в одной вершине этой же ветви:

$$\forall k: H(\beta k) = Ak0, \Psi ik0 \in Fr(Ai), Ai \in \beta 1 \rightarrow$$

$$\rightarrow \Psi kN \in T0(Aj), T(\Psi kN) = 3, Aj \in \beta 1 \qquad (2.7)$$

В ветви $\beta 1$ управление из вершины Ai после запуска ветвей βk, k = k1, k2, …, K, передается вершине Aj. Вершина Aj запускается на выполнение после завершения работы ветвей $\beta k1$ … βK. Таким образом, в каждый момент времени в любой ветви выполняется ровно одна вершина.

Переход между двумя вершинами, принадлежащими различным параллельным ветвям, возможен только по параллельным и терминирующим дугам, то есть, запрещены условные переходы между вершинами различных параллельных ветвей:

$$\forall i, j, \; Ai \in \beta 1, Aj \in \beta 2 \rightarrow T(\Psi ij) \neq 1 \qquad (2.8)$$

Если некоторая ветвь породила новые параллельные ветви, то вычисления в ней приостанавливаются до завершения работы порожденных ветвей. Таким образом, вложенные параллельные ветви выполняются последовательно относительно друг друга.

2.1.3. Модель памяти.

В технологии ГСП предполагается, что словарь данных, описывающих объекты предметной области располагается в общей памяти и одинаково доступен всем объектам граф-программы. Вместе с тем, большинство современных высокопроизводительных вычислительных систем, для которых разрабатываются программы в технологии ГСП, являются кластерами и имеют распределенную память. Чтобы обеспечить возможность разработки программ для таких систем, не обременяя пользователя технологии ГСП изучением механизмов обмена данными в распределенных системах, в технологии ГСП реализована модель общей памяти. Эта модель позволяет программе, разработанной в технологии ГСП, успешно выполняться в распределенной системе.

2.1.4. Граф-машина

В технологии ГСП для объектов–агрегатов используется централизованное управление процессом вычислений, осуществляемое специальной программой – *граф-машиной*.

Граф-машина универсальна для любого алгоритма. Исходной информацией для граф-машины служит описанная выше графическая модель алгоритма. Анализируя эту графическую модель, представленную в виде структур на смежной памяти, она выполняет в соответствующем порядке акторы и агрегаты, вычисляет значения предикатов и управляет синхронизацией. Для каждой параллельной ветви запускается по одному экземпляру граф-машины, которая представляет собой отдельный процесс в вычислительной системе.

Работа граф-машины начинается с выполнения актора в корневой вершине. Затем строится список дуг, исходящих из текущей вершины. Этот список просматривается граф-машиной последовательно, начиная с самой приоритетной дуги. Вычисляется значение предиката, помечающего дугу, и в случае его истинности, происходит переход к обработке следующей вершины.

В результате обработки параллельной дуги в отдельном процессе запускается другая граф-машина, обрабатывающая порождаемую данной дугой параллельную ветвь. После запуска всех параллельных ветвей происходит переход в вершину, в которой они терминируются. Родительская граф-машина ожидает завершения выполнения всех дочерних граф-машин, если не задано альтернативное условие.

Централизация функций управления в рамках одной программы (граф-машины) позволяет:

– контролировать вычислительный процесс в целом и в случае нештатных ситуаций принимать системные решения;

– реализовать сбор статистической информации о характеристиках надежности каждого из модулей, вычислительной сложности модулей, маршрутах развития вычислительного процесса и т.п.

2.2. Данные и объекты средства PGRAPH

Создание модели алгоритма в системе PGRAPH начинается с изучения предметной области (ПО), в которой будут производиться вычисления. Предметная область описывается совокупностью набора данных и набора операторов (вычислительных модулей), производящих вычисления над этими данными (словарь данных и библиотека вычислительных модулей) (см. рисунок 2.3).

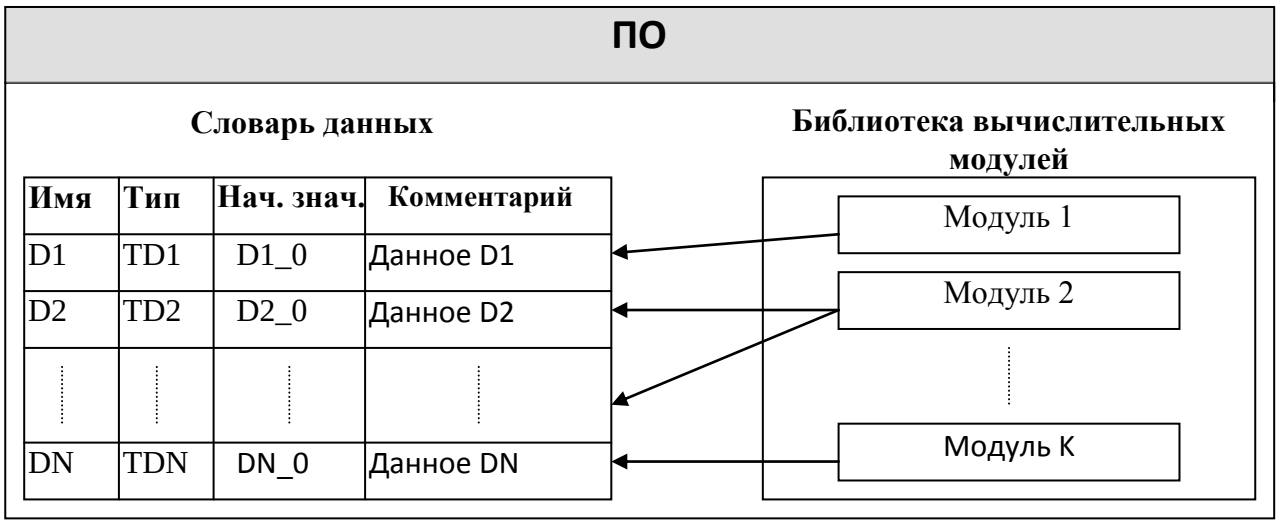


Рисунок 2.3 – Предметная область

Словарь данных представляет собой таблицу, в которой для каждого данного определено уникальное имя, тип, начальное значение и краткий комментарий к его назначению в ПО.

Вычислительные модули делятся на базовые модули и объекты. Под базовым модулем понимается исходный текст вычислительного модуля, представленный в виде функции на одном из языков программирования (например, C или C++), с соблюдением определенного формата.

Характерная форма модуля — это текстовый файл, являющийся объектом для текстового редактора. Удобство редактирования и наглядность диктуют ограничения сверху на размер модуля, который обычно колеблется в диапазоне 30-300 строк исходного текста.

Базовый модуль на уровне интерфейса оперирует формальными параметрами и описывает алгоритм преобразования формальных параметров. В предлагаемой модели понятие формального параметра заменено понятием типа данного, поскольку на самом деле другие атрибуты параметров в базовых модулях не несут никакой смысловой нагрузки, важен лишь тип параметра и порядок его использования, а “осмысление” назначения параметров возникает только после их аппликации к фактическим параметрам. Например, процедура, реализующая формулу A = B*C, в одной интерпретации типов данных вычисляет силу F по заданным ускорению *a* и массе *m* материальной точки (F = *a*m)*, в другой - путь S, по заданной скорости V и времени t (S = V*t).

Формально базовый модуль реализует отображение типов данных из области определения на область значений: $B_k : T^{in} \to T^{out}$, где $T^{in} = \langle t_1^{in}, t_2^{in}, \ldots, t_n^{in} \rangle$ — множество входных типов данных базового модуля B_k, $T^{out} = \langle t_1^{out}, t_2^{out}, \ldots, t_m^{out} \rangle$ — множество выходных типов данных.

На основе базового модуля создаются объекты модели, называемые

акторами. Один базовый модуль может породить множество объектов.

Под актором понимается специальным образом построенный вычислительный модуль, выполняющий определенные действия над данными ПО.

Акторы производят те же действия, что и породивший их базовый модуль, но над конкретными данными ПО (см. рисунок 2.4). Таким образом, акторы представляют собой отображение над множеством данных предметной области: $A_k : D_k^{in} \to D_k^{out}$, где $D_k^{in} = \{d_1^{in}, d_2^{in}, \ldots, d_n^{in}\}$ — множество входных данных актора A_k, $D_k^{out} = \{d_1^{out}, d_2^{out}, \ldots, d_m^{out}\}$ — множество выходных данных актора A_k. Множества D^{in} и D^{out} образуют в совокупности полное множество данных некоторой предметной области (словарь данных): $D = D^{in} \cup D^{out}$.

Соответствие между базовым модулем B_i и актором A_j порождает соответствие между подмножеством типов T_i и подмножеством данных предметной области D_j:

$$\begin{cases} B_i(T_i^{in}, T_i^{out}) \sim A_j(D_j^{in}, D_j^{out}) \\ T_i = (T_i^{in}, T_i^{out}) \sim D_j = (D_j^{in}, D_j^{out}) \end{cases}$$

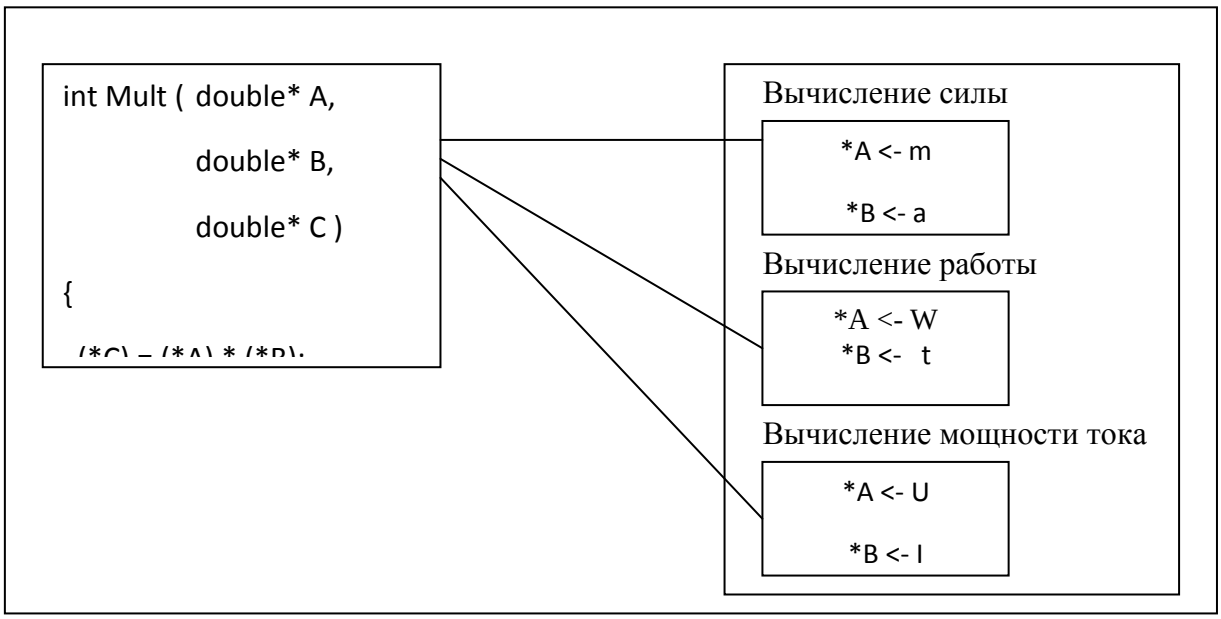


Рисунок 2.4 – Базовый модуль и акторы

Привязка объектов к данным ПО производится путем формирования так называемого паспорта объекта. Процедура привязки называется паспортизацией объекта.

Под паспортом объекта понимается таблица, содержащая перечень имен формальных параметров и соответствующих им имен данных ПО с указанием способа получения ими своих значений. По способу получения своих значений данные в паспорте делятся на три группы:

1) инициируемые (импортируемые) данные, которые должны принять значения до их использования объектом;

2) вычисляемые (экспортируемые) данные, которые впервые получают свои значения в процессе выполнения объекта (в процессе вычислений);

3) множества инициируемых и вычисляемых данных могут пересекаться, образуя множество модифицируемых (изменяемых) данных.

Таким образом, создание объекта на основе базового модуля сводится к формированию паспорта объекта. Процесс паспортизации заключается в установлении соответствий между именами формальных параметров базового модуля и именами данных ПО. При этом необходимо следить за совпадением типов формальных параметров базового модуля и соответствующих данных ПО.

На основе паспорта между базовым модулем и актором осуществляется односторонняя связь типа “один ко многим”. Каждый актор имеет свой прототип в виде базового модуля, а на основании каждого базового модуля можно построить один или несколько акторов (рисунок 2.5):

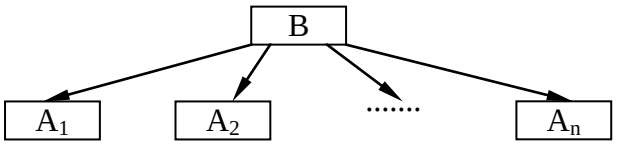


Рисунок 2.5 – Построение акторов на основе базового модуля

Это свойство полиморфизма объектов позволяет избежать избыточности

при порождении новых акторов, которые различаются между собой только привязкой по данным. Другими словами, на основе одного отлаженного и оттестированного базового модуля за счет механизма автоматизированной привязки по данным можно построить несколько корректных акторов, что позволяет значительно ускорить создание модели.

Кроме акторов используются еще два вида объектов: **предикаты** и **агрегаты**. Отличие между объектами заключается в способе использования данных. Акторы и агрегаты могут изменять значения данных, с которыми они работают, а предикаты не изменяют значений данных. При этом акторы и агрегаты вырабатывают признак аварийного или нормального завершения вычислений, а предикаты — признак истинности или ложности проверки некоторого условия над значением данных.

Формально предикат представляет собой отображение из множества данных предметной области на множество логических значений “истина” и “ложь”: Ψ_k: $(d_1, d_2, \ldots, d_m) \rightarrow \{0, 1\}$. Невозможность изменения предикатом данных, с которыми он работает, делает все его данные входными: $(d_1, d_2, \ldots, d_m) \in D^{in}$.

Объекты являются исходным материалом для построения графической модели - агрегата. Агрегат создается в форме графа, в котором объекты ПО играют роль вершин и дуг. Дуги — предикаты, а вершины — акторы или агрегаты. Дуги графа определяют передачу управления от одной вершины к другой.

Предикат — это логическая функция, которая в зависимости от значений данных ПО равна 0 или 1. Если значение 0, то переход по дуге запрещен. Иначе — переход разрешен. Переход выполняется после завершения вычислений в текущей вершине, по самой приоритетной из всех разрешенных дуг, исходящих из нее.

Формально агрегат представляет собой помеченный ориентированный граф с входной (корневой) и выходной (концевой) вершинами: $G = \{A, \Psi, \Phi, R\}$, где $A = \{A_1, A_2, \ldots, A_n\}$ – множество вершин графа, каждая

вершина A_i помечена некоторым актором, $\Psi = \{\Psi_{1i1}, \Psi_{1i2}, \ldots, \Psi_{jm}\}$ - множество дуг управления, $\Phi = \{\Phi_{1i1}, \Phi_{1i2}, \ldots, \Phi_{jl}\}$ - множество дуг синхронизации, R — отношение над множествами вершин и дуг графа, определяющее способ их связи.

Корневой вершиной графа A_0 является такая вершина, из которой есть маршрут по графу в любую другую вершину, и которая помечена как корневая. С этой вершины начинается выполнение алгоритма, описываемого агрегатом. Аналогично определяется концевая вершина A_n — это вершина, в которую есть маршрут из любой другой вершины, и которая не имеет исходящих дуг-предикатов. Концевых вершин на графе может быть несколько, если все они удовлетворяют поставленным условиям.

Развитие процесса вычислений в агрегате происходит путем передачи управления из одной вершины в другую, начиная с корневой. Этот процесс завершается, когда достигнута концевая вершина. Пример агрегата, построенного в системе моделирования и анализа алгоритмов параллельных вычислений PGRAPH, приведен на рисунке 2.6.

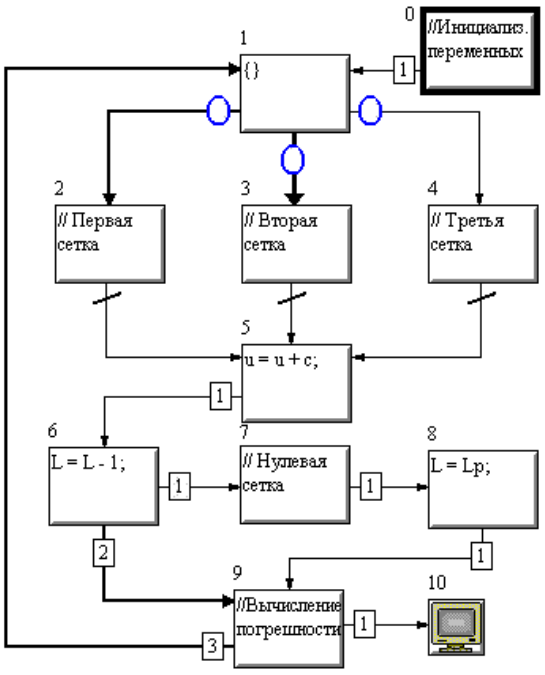


Рисунок 2.6 – Пример агрегата, описывающего параллельный алгоритм

2.3. Конструирование агрегатов в PGRAPH

2.3.1. Последовательные агрегаты

В наиболее общем виде разработка программы на основе графической модели алгоритма в технологии ГСП состоит из следующих этапов:

1. Создание словаря данных ПО. На данном этапе создаются новые типы и структуры данных, а также происходит накопление словаря данных, где хранится информация обо всех переменных программы.

2. Разработка базовых модулей. Это этап традиционного текстового программирования, на котором программист работает с исходными текстами программ, учитывая, однако, требования стандарта ГСП к оформлению этих текстов.

3. Создание объектов ПО. Этот этап производится автоматически после привязки формальных параметров базовых модулей к фактическим данным предметной области. Объекты, формируемые на основе базовых модулей вычислительного характера, называются акторами и формируют множество операторов F модели ГСП.

4. Конструирование агрегатов. На этапе графического программирования пользователь может создать графовый образ новой программы.

Разработанный и отлаженный агрегат, в свою очередь, может быть использован в качестве исходного материала при конструировании следующих агрегатов. Таким образом, в общем случае агрегат имеет иерархическую структуру.

Эффективность программирования в технологии ГСП возрастает по мере развития пользователем своей среды программирования. Доля текстового программирования с традиционной трудоемкой отладкой постепенно снижается, и программирование перерастает в конструирование агрегатов из надежных программных модулей. Отладка при этом заключается только в корректировке структуры графа.

2.3.2. Параллельные агрегаты

2.4. Методы синхронизации параллельных вычислений в PGRAPH

При создании математических моделей параллельных вычислений ключевой проблемой является синхронизация вычислений, т.е. организация их согласованного выполнения. Существуют различные способы синхронизации, такие как критические интервалы, семафоры, обмен сообщениями и почтовые ящики, мониторы /33/. В предлагаемой модели применяется комбинированный способ, использующий одновременно механизмы передачи сообщений и принципы мониторной синхронизации.

В графическую модель алгоритма вводятся дуги нового вида: дуги синхронизации $\Phi = \{\Phi_{1i1}, \Phi_{1i2}, \ldots, \Phi_{jl}\}$.

Каждая дуга синхронизации Φ_{ij} помечена сообщением $\mu_{ij} \in N$. В классической (последовательной) модели алгоритма, используемой в технологии ГСП, отсутствуют дуги синхронизации, которые вводятся в модель параллельного алгоритма для решения задач синхронизации между его различными участками.

Определим почтовый ящик Lpost как список, формируемый из сообщений, с помощью которых вершины модели информируют друг друга о своем состоянии: Lpost = [μi0,j0, …, μim,jn], где μi,j – сообщение, посылаемое от вершины Ai вершине Aj. Возможность передачи сообщения от одной вершины граф-модели другой вершине изображается графически дугой синхронизации Фij, проведенной из вершины-источника сообщения в вершину-получатель сообщения. Дуги синхронизации изображаются пунктирными стрелками (рисунок 2.1, б).

Вершина Ak посылает сообщения другим вершинам, формируя список сообщений LAkCom = [μk,j0, …, μk,jn] и добавляя его к списку Lpost. Наличие сообщения μi,j в списке Lpost говорит о том, что оператор вершины Ai завершил работу и хочет сообщить об этом вершине Aj.

Каждой вершине ставится в соответствие семафорный предикат RAj = r (bi0,j, …, bim,j), представляющий собой правильно построенную формулу относительно булевых переменных bik,j, связанных логическими связками типа Λ, V.

Булевы переменные определяются следующим образом:

$$b_{k,j} = \begin{cases} 1, \text{ если } \mu_{k,j} \in Lpost, \\ 0, \text{ если } \mu_{k,j} \notin Lpost. \end{cases}$$

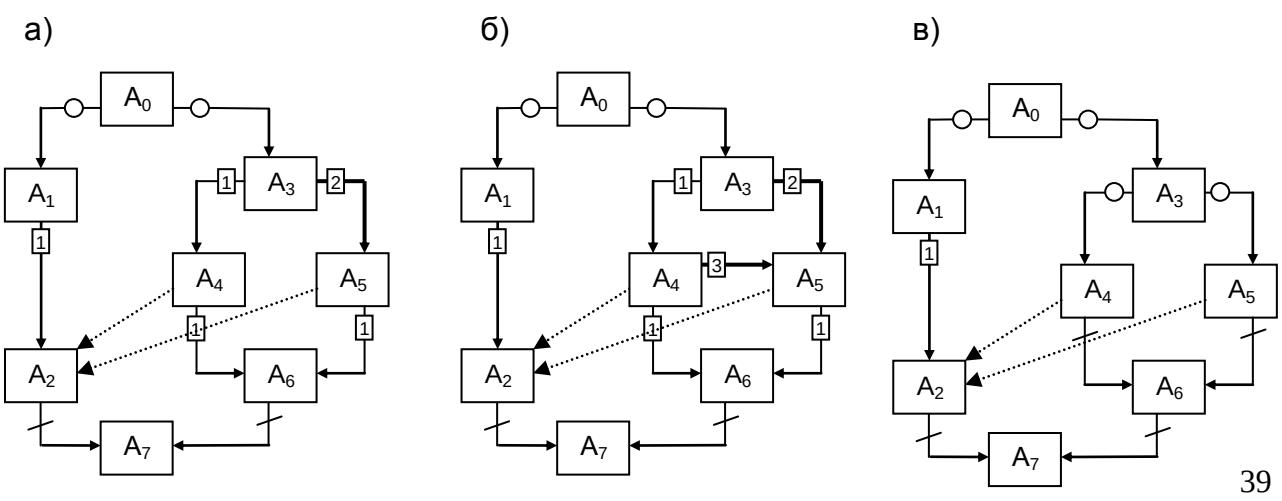
Семафорный предикат разрешает или запрещает запуск соответствующей вершины. Если семафорный предикат RAj = 1, то запуск оператора вершины Aj разрешается. В противном случае вычисления приостанавливаются до того момента, когда RAj станет равным 1. Семафорный предикат определяется для всех вершин, в которые входят дуги синхронизации. Для остальных вершин значение семафорного предиката принимается тождественно истинным.

Обмен сообщениями между вершинами, принадлежащими одной параллельной ветви, запрещен:

$A_i \in \beta_k, A_j \in \beta_k \rightarrow b_{i,j} \equiv 0$

Вершины в пределах одной параллельной ветви исполняются строго последовательно, поэтому их синхронизировать не требуется.

Замечание Операцию "v" в семафорных предикатах следует использовать только над операндами, определенными над взаимоисключающими вершинами, то есть такими вершинами, факт запуска



одной из которых исключает получение управления остальными вершинами. На рисунке 2.2, а) вершины А4 и А5 являются взаимоисключающими.

Рисунок 2.2 – Пример использования операции "v" в семафорных предикатах

Если вершины А2, А4 и А5 используют общее данное, а операторы в вершинах А2, А3 и А6 имеют относительно большую длительность исполнения, то имеет смысл ввести две дуги синхронизации Ф4,2 и Ф5,2 и определить семафорный предикат для вершины А2 с использованием операции "v": RA2 = b4,2 v b5,2. В случае передачи управления от вершины А3 к одной из вершин А4 или А5 (например, вершине А4), другая вершина (А5) уже не запустится, а значит, вершину А2 достаточно синхронизировать только с вершиной, получившей управление (А4). В этом случае для запуска на исполнение операнда в вершине А2 достаточно истинности одного из операндов ее семафорного предиката, и можно использовать операцию "v".

Если вершины не являются взаимоисключающими, то использование операции "v" над их сообщениями не разрешает конфликта совместного использования данных, так как приход сообщения от одной из таких вершин не исключает запуска другой на исполнение. Примеры подобных ситуаций приведены на рисунке 2.2, б), в).

На рисунке 2.2, б) граф-модель дополнена дугой Ψ4,5. Если управление передается по этой дуге от вершины А4 к вершине А5, то оператор последней может исполняться одновременно с оператором вершины А2, что приведет к конфликту совместного использования данных.

На рисунке 2.2, в) вершины А4 и А5 принадлежат различным параллельным ветвям. В этом случае использование операции "v" в семафорном предикате вершины А2 также не исключает конфликта, поскольку операторы вершин А4 и А5 могут исполняться одновременно, и приход сообщения от одной из них не гарантирует, что оператор другой вершины также завершил исполнение.

Приведенные рассуждения и примеры показывают, что операцию "v" в семафорных предикатах следует применять с осторожностью, и только в тех случаях, когда легко видеть, что вершины, от которых исходят дуги синхронизации, являются взаимоисключающими.

3. Пользовательский интерфейс средства PGRAPH

3.1. Установка системы PGRAPH. Требования к программному обеспечению.

3.2. Начало работы с PGRAPH. Рабочее пространство PGRAPH.

3.3. Создание новой предметной области моделирования (ПрОМ).

3.4. Выбор и открытие предметной области моделирования.

3.5. Описание структур данных ПрОМ в PGRAPH.

3.6. Словарь ПрОМ. Работа со словарем. Описание данных ПрОМ.

В соответствии с методологией ГСП, разработка программы начинается с определения используемых типов данных и словаря данных ПО. Эти действия выполняются в диалоговых окнах «Список типов» и «Словарь данных», вызываемых с помощью одноименных пунктов меню «Данные».

Для вновь создаваемого типа вводится его имя и определение в соответствии с синтаксисом языка C++. Базовые типы языка C++ уже содержатся в информационном фонде системы.

При создании данного (переменной) предметной области пользователь системы определяет:

- имя данного;
- тип данного (выбирается из списка существующих в предметной области типов);
- начальное значение;
- комментарий, описывающий назначение данного.

3.7. Создание акторов в ПрОМ.

Граф-программа строится из объектов технологии ГСП: акторов и предикатов.

Для создания объектов ГСП служит «Редактор объектов», вызываемый выбором одноименного пункта в меню «Объект» (рисунок 2.4).

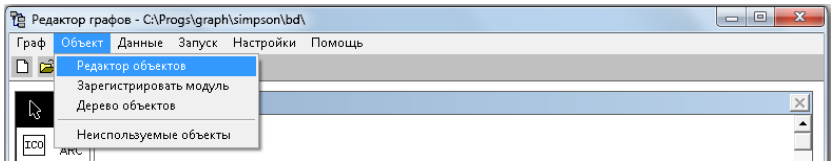
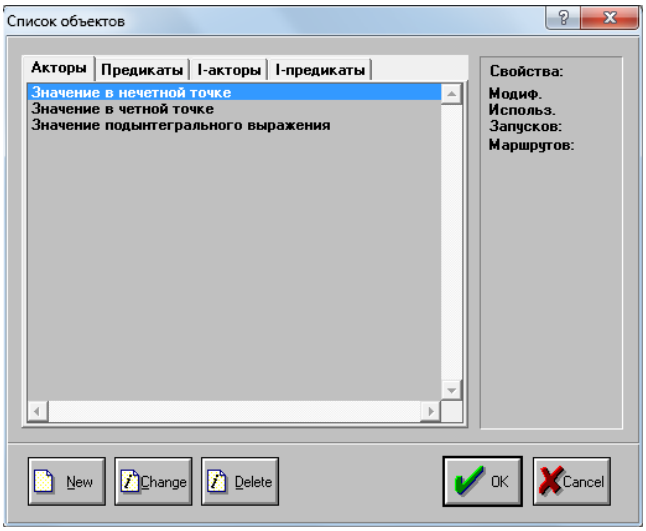


Рис. 2.4 – Вызов редактора объектов



Окно редактора объектов приведено на рисунке 2.5.

Рис. 2.5 – Окно редактора объектов

Объекты ГСП условно делятся на две группы: Inline-объекты (I-акторы, I-предикаты) и обычные объекты (Акторы и Предикаты).

Обычные объекты создаются на основе базовых модулей. Inline-объекты отличаются от обычных тем, что их базовые модули создаются непосредственно при создании самого объекта. Как правило, inline-объекты выполняют простые действия. Например, I-актор может выполнять инкремент счетчика (J = J + 1;).

В этом случае для него нецелесообразно создавать отдельный файл с базовым модулем. Вместо этого текст базового модуля вводится непосредственно при создании I-актора. Аналогично, несложные предикаты,

осуществляющие например, сравнение двух переменных, оформляются в виде I-предикатов.

Для создания актора, предиката, I-актора или I-предиката необходимо выбрать соответствующую вкладку в окне Редактора объектов и нажать кнопку «Создать».

3.7.1. Базовые модули. Регистрация базовых модулей в ПрОМ.

Базовые модули служат основой для построения объектов технологии ГСП (акторов и предикатов).

Исходный текст базового модуля создается во внешнем текстовом редакторе и сохраняется в виде файла с расширением ".C" в рабочий каталог системы PGRAPH.

Для использования базового модуля его необходимо зарегистрировать в системе. Регистрация производится путем выбора меню "Объект" -> "Зарегистрировать модуль" (рисунок 2.2).

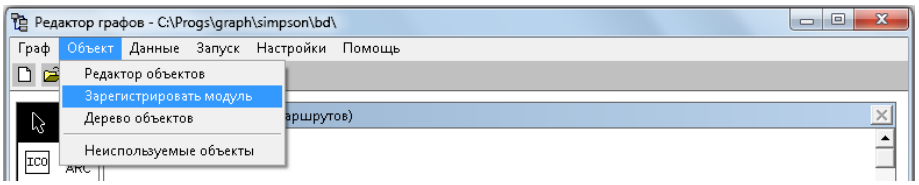


Рис. 2.2 – Меню регистрации базового модуля

Диалоговое окно регистрации базового модуля изображено на рисунке 2.3. Процесс регистрации состоит из двух шагов. На первом шаге нужно выбрать файл с исходным текстом базового модуля. В поле «Комментарий» вводится назначение и краткое описание создаваемого базового модуля.

На втором шаге для каждого параметра базового модуля необходимо определить его класс (Исходный, Вычисляемый или Модифицируемый).

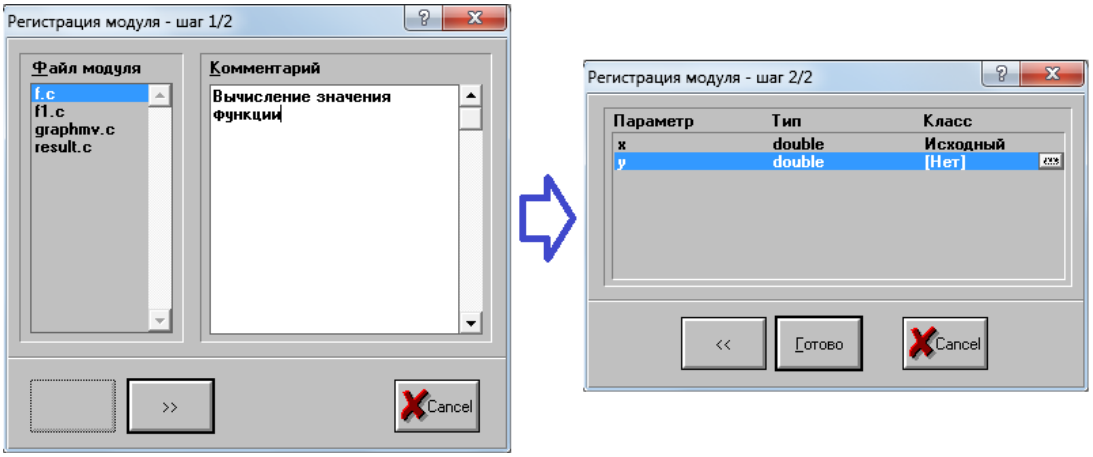


Рис. 2.3 - Диалоговое окно регистрации базового модуля

После нажатия на кнопку «Готово» базовый модуль регистрируется в системе.

3.7.2. Inline-модули. Создание inline-модулей в ПрОМ.

Создание I-актора состоит из двух шагов, приведенных на рисунках 2.6 и 2.7.

На первом шаге (рисунок 2.6) вводится исходный текст I-актора и выбирается картинка (иконка), которая может отображаться на вершинах с этим I-актором вместо текста.

Переход ко второму шагу происходит по нажатию на кнопку " >> ".

На втором шаге для каждого данного ПОП, используемого в I-акторе, назначается его класс (исходное, вычисляемое или модифицируемое) и вводится краткий комментарий с описанием этого данного (рисунок 2.7).

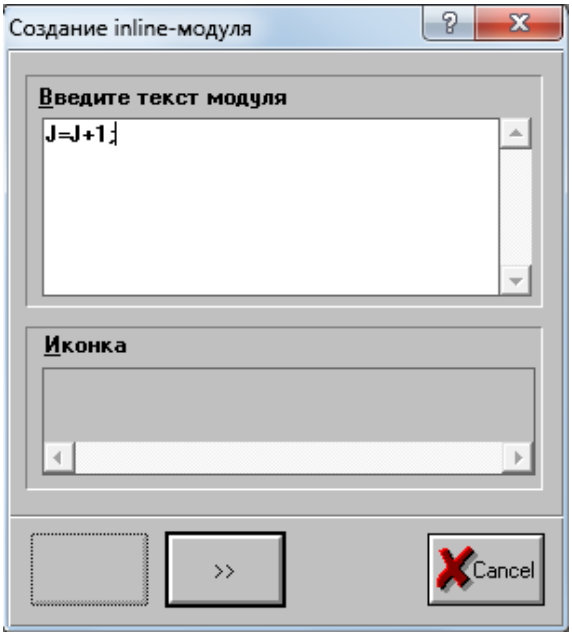


Рис. 2.6 – Окно редактирования исходного текста inline-актора

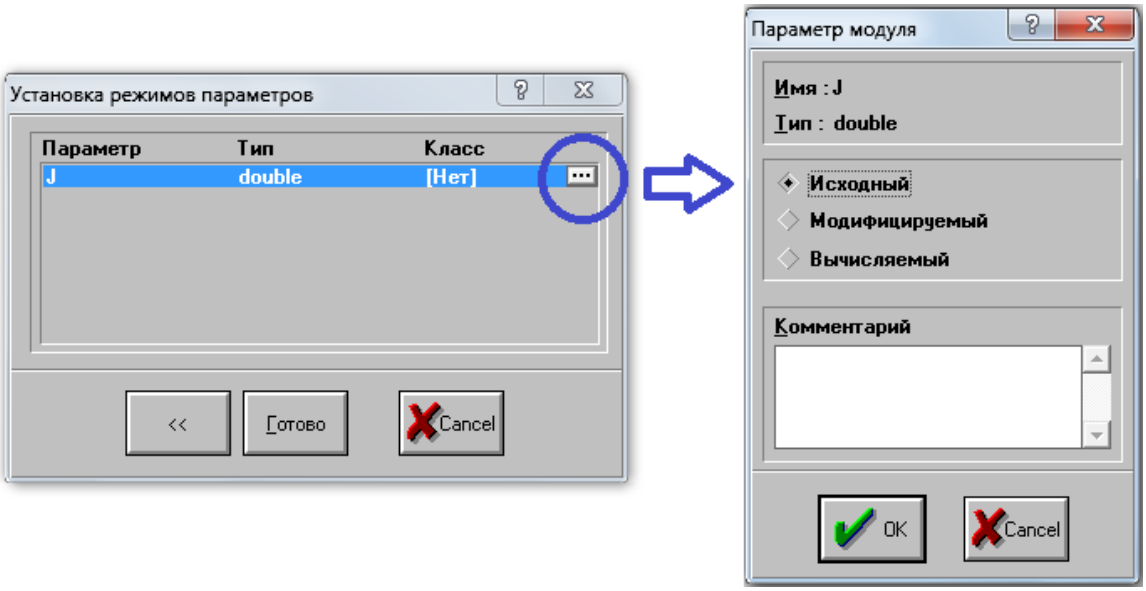


Рис. 2.7 – Назначение класса использования для данных, используемых актором

Создание inline-предикатов.

Создать I-предикат очень просто. Достаточно ввести его исходный текст (т.е. логическое условие над данными ПОП) и нажать кнопку «ОК» (рисунок 2.8).

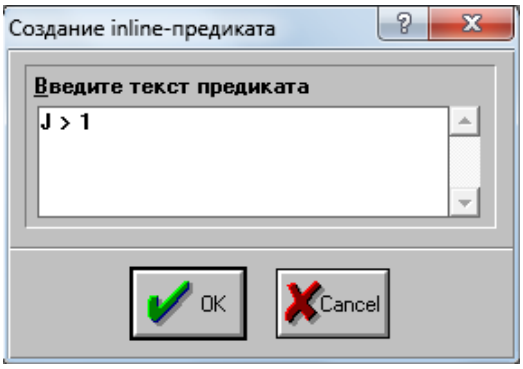


Рис. 2.8 – Создание inline-предиката

3.7.3. Описание акторов ПрОМ.

Акторы, в отличие от I-акторов, создаются на основе базовых модулей. Процесс создания актора состоит из трех шагов.

На первом шаге вводится имя для вновь создаваемого актора и выбирается картинка (иконка) для отображения на вершинах в этим актором (рисунок 2.9).

На втором шаге выбирается базовый модуль, на основе которого строится актор (рисунок 2.10).

На третьем шаге производится паспортизация базового модуля: каждому его параметру ставится в соответствие данное ПОП такого же типа (рисунок 2.11).

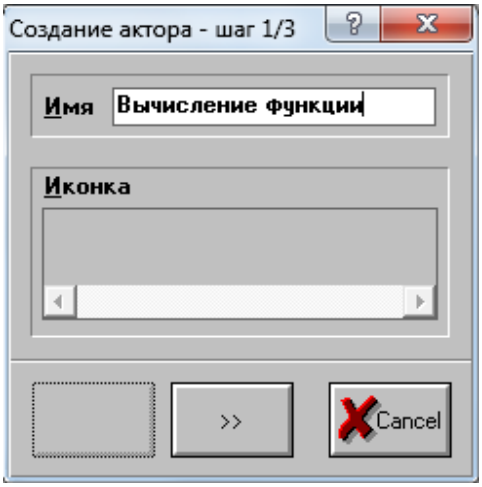


Рис. 2.9 – Первый шаг создания актора

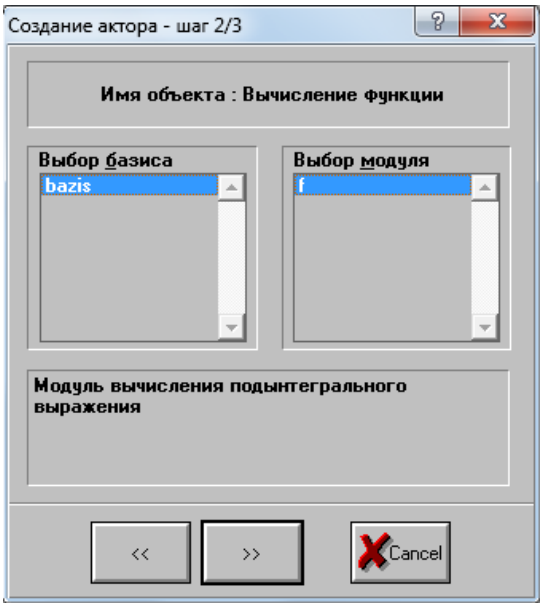


Рис. 2.10 – Второй шаг создания актора

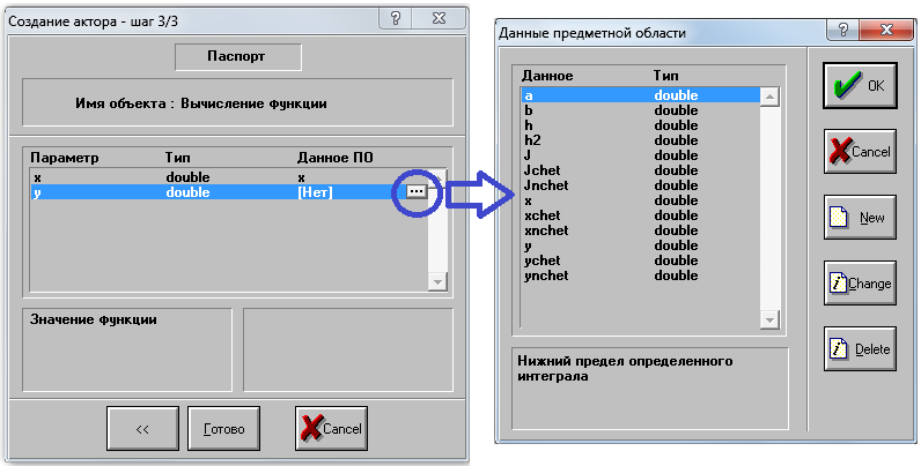


Рис. 2.11 – Третий шаг создания актора (паспортизация)

Процесс создания предиката аналогичен процессу создания актора. Он также состоит из трех шагов: ввод имени предиката, выбор базового модуля, паспортизация.

3.7.4. Прикрепление иконок к акторам ПрОМ.

3.8. Графический редактор PGRAPH.

3.8.1. Рабочее поле редактора.

3.8.2. Основные инструменты редактора.

3.8.2.1. Формирование вершины управляющего графа модели алгоритма. Начальная вершина графа.

3.8.2.2. Посторенние дуги графа модели алгоритма, Задание приоритетов дугам графа

3.8.2.3. Удаление вершин и дуг графа..

3.8.2.4. Редактирование графа модели алгоритма

3.8.2.5. Привязка вершин графа к объектам ПрОМ.

3.8.2.6. Привязка дуг графа к предикатам ПрОМ.

3.8.2.7. Сохранение управляющего графа модели алгоритма в ПрОМ. Создание агрегата ПрОМ.

Список литературы

Основная литература

1. Коварцев А.Н., Жидченко В.В. Методы и средства визуального параллельного программирования. Автоматизация программирования: учеб. – Самара: Изд-во Самар. Гос. Аэрокосм. Ун-та, 2011.- 168 с.

Дополнительная литература

1. Коварцев А.Н. Автоматизация разработки и тестирования программных средств. - Самарский государственный аэрокосмический университет., Самара, 1999. – 150 с.