ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «ОРЕНБУРГСКИЙ ГОСУДАРСТВЕННЫЙ АГРАРНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ МАТЕРИАЛЫ ДЛЯ ОБУЧАЮЩИХСЯ ПО ОСВОЕНИЮ ДИСЦИПЛИНЫ

Б1.В.06 Современные проблемы информатики и вычислительной техники

Направление подготовки (специальность)
09.04.01 Информатика и вычислительная техника

Профиль подготовки (специализация)
“Автоматизированные системы обработки информации и управления**”**

Форма обучения *очная*

СОДЕРЖАНИЕ

1. Тематическое содержание дисциплины

Раздел 1. Взаимосвязь между информационными технологиями и решаемыми прикладными задачами

1.1. Тема 1: Общие особенности процесса разработки программного обеспечения. Методические приемы. Формализация предметной области (4 часа)

1.1.1. Перечень и краткое содержание рассматриваемых вопросов:

1. Общие особенности процесса разработки программного обеспечения

Среди множества факторов, от которых зависит процесс разработки программного обеспечения, можно выделить:

– классы решаемых, задач, определяющие смысловое содержание создаваемых программ;

– методологии, задающие особенности организационного и технического проведения основных этапов разработки программного обеспечения;

– методы и парадигмы программирования, обуславливающих стили кодирования и архитектуры виртуальных машин;

– аппаратные и системные программные средства, предоставляющие логические и физические ресурсы для непосредственного использования ПО.

Разнообразие этих факторов определяет множество вариантов, связанных с организацией процесса разработки.

Цель процесса разработки – создание программы, обеспечивающей решение поставленной задачи некоторым исполнителем. Решаемая задача описывается совокупностью формальных и эмпирических (неформальных) моделей, определяющих как протекающие процессы, так и используемые при этом данные.

2. Методические приемы

Методические приемы ориентированы на формализацию представления моделей и методов перехода между ними. Они позволяют ускорить процесс разработки следующими способами:

– формализацией предметных областей;

– созданием методик разработки программного обеспечения.

3. Формализация предметной области

Формализация предметной области заключается в построении ее модели и разработке методов преобразования модели предметной области в модель исполни-теля. Модель предметной области объединяет совокупность специализированных моделей предназначенных для описания определенного класса решаемых задач, что обеспечивают унификацию решения сверху. Дальнейший переход к модели исполнителя обычно осуществляется по выработанным методам или алгоритмам.

Подобный подход широко используется при разработке программ в разных предметных областях. В качестве примеров, можно привести:

1) Построение синтаксических и лексических анализаторов. Модель предметной области описывается с использованием формальных грамматик, определяющих принципы порождения правильных цепочек. Наличие эквивалентности между различными грамматиками и автоматами позволяет перейти к распознавателям цепочек путем использования наработанных методов их программной реализации.

2) Разработка программных систем на основе теории автоматов. Последующая программная реализация автоматов является хорошо отработанным формальным приемом с применением различных технологий.

Разработка алгоритмов преобразования одних моделей в другие позволяет автоматизировать процесс и обеспечить представление исходной задачи в виде формализованных данных или программы на специализированном (проблемно-ориентированном) языке программирования. Фактически это означает слияние моделей задачи и исполнителя. По такой схеме разрабатываются специализированные языки и системы программирования. К ним, в частности, можно – отнести:

– многие языки имитационного моделирования;

– языки, используемые для автоматного программирования, например, язык SDL.

– системы автоматизированной генерации лексических и синтаксических анализаторов языков программирования по описанию синтаксиса на соответствующих метаязыках.

1.2. Тема 2: Создание методик разработки программного обеспечения (6 часов).

1.2.1 Перечень и краткое содержание рассматриваемых вопросов:

Методики разработки программного обеспечения ориентированы на формализацию взаимосвязей между моделями конкретных исполнителей и моделями, используемыми на предшествующих этапах разработки (например, при анализе и проектировании).

Место методик (состоящих из набора методов преобразования между различными группами моделей) в процессе разработки ПО представлено на рис. 1.

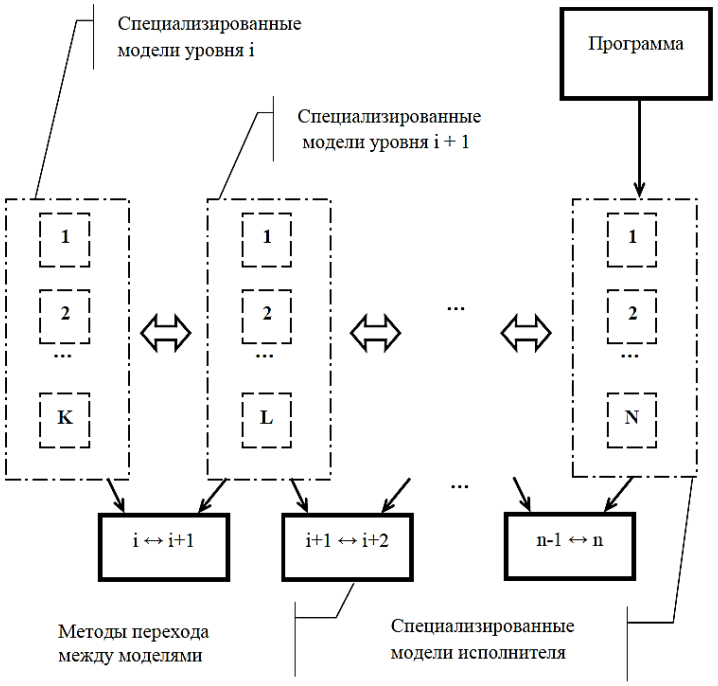

Рис. 1. Использование методик разработки для поэтапного преобразования моделей

Изначальная ориентация Исполнителя на универсальность вычислений обычно предполагает применение методик для широкого класса задач, не связывая их непосредственно с предметными областями. Они поддерживают взаимодействие моделей, используемых в разработке, определяя процесс преобразования как от модели задачи к модели исполнителя, так и в обратном направлении.

Несмотря на обеспечение прямого и обратного проектирования, *основное достоинство методик* проявляется в поддержке нисходящей разработки, что во многом обуславливается большей наглядностью переходов от универсальных высокоуровневых

моделей, ориентированных на описание предметных областей, к моделям, описывающим соответствующих исполнителей.

В подобной ситуации знание задачи повышает эффективность разработки. Поэтому, создание программного обеспечения обычно начинается с привязки модели предметной области к моделям анализа и проектирования, предлагаемым используемой методикой (рис. 2).

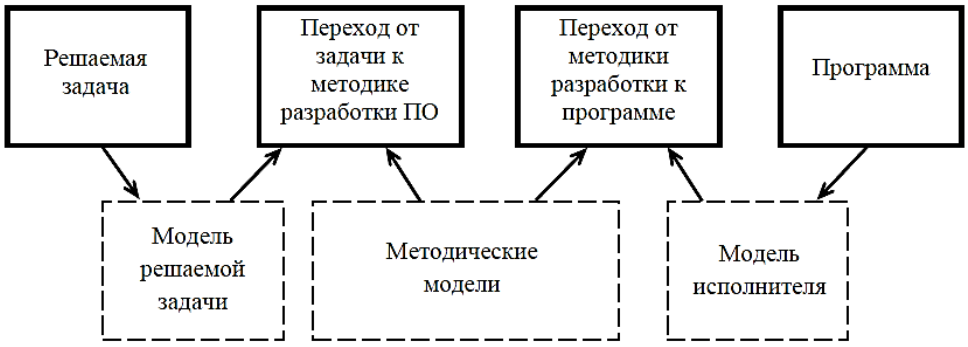

Рис. 2. Совместное использование методов формализации предметной области и методик разработки

К достоинствам методик разработки ПО следует отнести:

- универсальность, обуславливающая ориентацию на разработку задач широкого класса;
- поддержку нисходящего и восходящего проектирования;
- поддержку прямого и обратного проектирования;
- возможность использовать инструментальные средства.

К недостатку отдельных методик относится привязка процесса разработки к определенным методам и исполнителям.

В настоящее время существуют различные методики. Они являются составной неотъемлемой частью методологий разработки программного обеспечения, которые, наряду с процессами создания программ, дополнительно регламентируют организационную деятельность, анализ, тестирование и сопровождение, что в целом определяет организацию жизненного цикла программы на основе единого концептуального подхода. Примерами подобных методик могут служить:

- объектно-ориентированный подход, используемый в составе объектно-ориентированной методологии;
- методы структурного анализа и проектирования, применяемые при разработке информационных систем;
- методы быстрой разработки приложений, ориентированные на построение программ от моделей, определяющих взаимодействие системы с пользователем.

Методики широко используются при разработке больших программных систем, так как повышают эффективность проектирования за счет предоставления достаточно простых и ясных подходов, выработанных на основе эмпирического опыта и теоретических исследований.

Их использование, в сочетании с инструментальной поддержкой, обеспечивает сокращение семантического разрыва между моделями различных задач и исполнителем.

1.3. Тема 3: Технические приемы. Поддержка методических приемов. Вспомогательные средства. Системы программирования (6 часов).

1.3.1. Перечень и краткое содержание рассматриваемых вопросов:

1. Технические приемы

Технические приемы обеспечивают создание инструментальных средств, поддерживающих различные аспекты разработки ПО. Можно выделить:

– средства поддержки методических приемов;

– вспомогательные средства;

– системы программирования.

2. Поддержка методических приемов

Инструментальные средства, обеспечивающие поддержку методических приемов, предназначены для компьютерного представления и анализа разрабатываемых моделей, преобразования построенных моделей в другие модели, автоматизации процесса построения требуемой документации, а также ведения организационной деятельности в ходе разработки ПО.

Повышая эффективность процесса разработки, средства поддержки методических приемов, в то же время, не определяют сам процесс построения программы. Их использование предполагает дальнейшую доводку программ с применением инструментов, имеющих более тесную связь с архитектурами вычислительных систем.

Примерами ранних средств подобного рода могут служить системы поддержки спецификаций. Во многих из них использовались графические языки, которые в основном играли роль вспомогательного документа. Для написания программы необходимо было вручную осуществлять перевод этих диаграмм в код.

Положение изменилось с внедрением CASE-средств. Разработанные инструменты сгладили семантический разрыв между рядом моделей предметной области и кодом, обеспечив непосредственное преобразование, как в прямом, так и в обратном направлении.

К средствам поддержки методов структурного анализа и проектирования можно отнести:

- BPwin, ERwin, ориентированные на использование диаграмм IDEF и DFD;

- UML, широко поддерживающий объектно-ориентированную и другие методологии.

3 Вспомогательные средства

Вспомогательные средства предназначены для повышения эффективности процессов, не связанных с непосредственной разработкой структуры программы, но, в то же время, сильно влияющих на качество и время разработки.

К ним следует отнести:

- средства отладки;

- средства профилирования и другие.

Их специфика заключается в работе с уже готовыми программами, что позволяет в дальнейшем игнорировать их влияние на сам процесс разработки мобильных и эволюционно расширяемых параллельных программ.

Отладка используется для локализации и устранения ошибок в разрабатываемых программах. Существуют различные подходы к решению этой задачи.

В частности проводятся следующие мероприятия:

- анализируются текущие значения различных переменных;

- определяются траектории выполнения программы.

Эти задачи можно решать путем вставки в исходные тексты программ специальных вспомогательных операторов. Однако вместо этого часто используются специальные отладчики, которые повышают эффективность процесса отладки за счет дополнительных сервисных функций.

Профилирование направлено на сбор и анализ характеристик, определяющих особенности функционирования программы.

К ним относятся:

- время выполнения отдельных фрагментов кода (подпрограмм, функций, потоков, процессов);

- количество истинных и ложных переходов в различных условных операторах и операторах цикла;

- количество кэш промахов и т.д.

Инструмент, предназначенный для получения этих характеристик, называется профилировщиком. Полученные данные используются для оптимизации программы повышающей эффективность ее функционирования, которая обычно связана с улучшением таких критериев качества как уменьшение времени выполнения программы, уменьшение объема занимаемой памяти и рядом других.

4. Системы программирования

Системы программирования – это инструментальные средства, поддерживающие разработку программ для заданного виртуального исполнителя и их последующее автоматическое преобразование в программы реального исполнителя.

Использование систем программирования позволяет решать задачу повышения эффективности процесса разработки ПО за счет сокрытия исполнителей более низкого уровня, являющихся реальными вычислительными системами. Та-кой подход широко используется на практике и позволяет писать программы для высоко-уровневого исполнителя (определяемого часто как архитектура виртуальной машины). Реальное выполнение полученных программ может осуществляться:

1) Использованием непосредственной интерпретации, полностью скрывающей процесс реального выполнения, который может оказаться намного сложнее.

2) Автоматическим и поэтапным преобразованием программы, написанной для виртуальной машины, в программу конечного исполнителя, обеспечивающего ее интерпретацию, что позволяет разрабатывать программы для систем, не допускающих непосредственное выполнение.

1.4. Тема 4: Характеристики систем программирования. Разделение систем программирования по парадигмам (4 часа).

1.4.1. Перечень и краткое содержание рассматриваемых вопросов:

1. Характеристики систем программирования

Системы программирования являются основным инструментом, используемым для написания программ, что определяет их значимость в достижении критериев качества.

Анализ организации систем программирования позволяет выделить *специфику инструментальных средств*, поддерживающих разработку программных систем.

2. Разделение систем программирования по парадигмам

Во многом особенности систем программирования определяются архитектурой исполнителей, разнообразие которых порождает множество подходов к решению даже одной и той же задачи. Это позволяет использовать различные комбинации специализированных моделей и ведет к появлению разных стилей программирования, определяемых также как парадигмы.

Парадигмы занимают важное место в технологии разработки программного обеспечения. Вокруг них начинают выстраиваться и развиваться методологические концепции.

Такая роль обуславливается тем, что возникающие новые идеи по созданию программ первоначально реализуются в простых инструментах, поддерживающих исследование и экспериментальную проверку выдвигаемого стиля.

Чаще всего в качестве инструментов выступают языки программирования. Упомянутые исследования начинаются с написания простых программ. Лишь после обобщения первоначального опыта приходит понимание достоинств и недостатков, позволяющих перейти к формированию методологий, обеспечивающих использование парадигмы при разработке больших программных систем. Если разработанная парадигма не способна служить основой промышленной методологии, она отвергается или применяется в ограниченных масштабах.

Классификация систем программирования по парадигмам является одной из наиболее популярных. Она позволяет осуществить достаточно четкую градацию, опираясь на основные отличительные признаки.

Различают пять основных стилей программирования (табл. 1).

Таблица 1

Основные стили программирования

Название стиля	**Основополагающие абстракции**
Логико-ориентированный	Цели, часто выраженные в терминах исчисления предикатов
Ориентированный на правила	Правила "если-то"
Ориентированный на ограничения	Инвариантные отношения
Процедурно-ориентированный	Алгоритмы, абстрактные типы данных
Объектно-ориентированный	Классы и объекты

Для некоторых из представленных вариантов можно провести дополнительную градацию. В частности, процедурно-ориентированный стиль содержит императивную и функциональную парадигмы программирования.

1.5. Тема 5: Дополнительные характеристики парадигм программирования. Методы алгоритмизации. Методы управления вычислениями. Организация программных объектов (10 часов).

1.5.1. Перечень и краткое содержание рассматриваемых вопросов:

1. Дополнительные характеристики парадигм программирования

Независимо от используемых подходов, системы программирования обеспечивают техническую поддержку процесса разработки, который характеризуется использованием ряда общих принципов, обуславливаемых целью: создание программы, допускающей дальнейшее выполнение непосредственно или после цепочки формальных и автоматических преобразований.

Вместе с тем, классификация систем программирования, опирающаяся на парадигмы, не позволяет объективно и всесторонне оценить параметры, определяющие различные способы написания программ.

Это затрудняет:

- исследование критериев качества программного обеспечения;
- создание моделей для оценки различных характеристик систем программирования;

- разработку новых методов построения программ.

При исследовании разнообразных критериев качества программ нужно опираться не только на общие черты, определяемые парадигмами. Необходимо выделить характеристики, обеспечивающие независимый анализ требуемых характеристик. В этом случае формирование общего восприятии анализируемой системы может быть достигнуто путем комбинирования альтернативных критериев.

Подобное разбиение можно провести по следующим составляющим:

- методам алгоритмизации решаемой задачи;
- методам управления вычислениями;
- способам организации программных объектов и формированию отношений между ними;
- специфике процессов преобразования модели исполнителя, заданной системой программирования, в модель исполнителя, осуществляющего реальное выполнение программ.

2. Методы алгоритмизации

Построение алгоритма решаемой задачи является одной из основных за-дач программирования. Разнообразие методов алгоритмизации порождает множество поведенческих парадигм, каждая из которых характеризуется своей спецификой описания процесса обработки данных и управления вычислениями, например:

- императивный стиль определяет исполнителя на основе традиционной фон-неймановской архитектуры, непосредственно исполняющего команды заданные программистом (при этом команды явно связаны между собой только по управлению);
- автоматное программирование выделяет в качестве основы модель состояний, что ведет к абстрагированию на этапе разработки программы от безусловных управляющих связей и ускоряет построение общей схемы логического управления алгоритмом.
- функциональное программирование опирается на теорию рекурсивных функций и описывает процесс управления в виде отношений между функциями;
- программирование, управляемое потоками данных, как и функциональное программирование использует отношение между функциями и операциями, но ориентировано на выполнение операций по готовности данных;
- декларативный стиль опирается на логику исчисления предикатов, которая обуславливает организацию алгоритма, непосредственно не описывающую вычисления (используемая для его выполнения машина логического вы-вода первоначально осуществляет анализ правил, задающих различные отношения, выбирая среди них на выполнение те, которые соответствуют заданным условиям).

Каждый из представленных стилей определяет специфическую интерпретацию понятия программы и обуславливает конкретные методы и приемы построения алгоритмов, которые могут сильно отличаться даже для одной и той же решаемой задачи.

3. Методы управления вычислениями

Методы алгоритмизации решаемой задачи тесно связаны с походами к управлению вычислениями, разнообразии которых обуславливается способами распараллеливания и синхронизации вычислений. Управление может варьироваться от последовательного до задания максимального параллелизма. Оно может задаваться явно программистом или неявно, когда используются отношения между данными и автоматическое управление ресурсами.

Все разнообразие методов управления раскрывается как через архитектуры разработанных вычислительных систем, так и языков последовательного и параллельного программирования.

Используемые методы управления вычислениями сильно влияют на мобильность параллельных программ, так как именно они непосредственно отражают все ограничения,

присущие архитектуре, как реальной вычислительной системы, так и архитектуре виртуальной машины, определяющей особенности исполнителя конкретного языка программирования.

4. Организация программных объектов

Разработка больших программных систем изначально опиралась на различные методы декомпозиции. В ходе исторического развития были опробованы различные схемы компоновки структур данных и программного кода. Ряд их используется в программировании и в настоящее время.

Наиболее типичными примерами являются:

- модульная структура;
- абстрактные типы данных;
- процедуры (подпрограммы, функции);
- интерфейсы;
- классы, использующие наследование и виртуализацию.

В отличие от методов алгоритмизации, *методы организации программных объектов* не определяют процесс выполнения решаемой задачи. Однако они оказывают существенное влияние на конструктивные характеристики программных объектов, такие, как их повторное использование, эволюционное расширение в ходе добавления в программу новой функциональности.

Организация программных объектов оказывает существенную роль на разработку больших программных систем. Именно она обуславливает основные тенденции в современном программировании, в частности, популярность объектно-ориентированного подхода и его дальнейшее развитие.

Раздел 2. Архитектуры вычислительных систем и проблемы параллельного программирования

2.1. Тема 6: Проблемы организации вычислительных систем. Использование последовательных компьютеров. Специфика параллельных вычислений (4 часа).

2.1.1. Перечень и краткое содержание рассматриваемых вопросов:

1. Проблемы организации вычислительных систем

Проблемы разработки программного обеспечения (ПО) зависят не только от сложности решаемых прикладных задач и особенностей методологий. Они также связаны с *организацией вычислительных систем* (ВС), обеспечивающих непосредственное выполнение программ.

Существование разнообразных инструментов, создающих иллюзию высокоуровневой виртуальной машины, не может скрыть специфику реальных архитектур, которую приходится учитывать при программировании. Очень часто это делает написанный код непереносимым. Использование методов автоматического преобразования программ между разными архитектурами в настоящее время является слишком сложным и малоэффективным.

Программирование для различных ВС уже давно не является последовательным. Появились новые архитектуры, поддерживающие разнообразные методы *параллельного программирования*. Их наличие обуславливается многими причинами как технологического характера, так и необходимостью решения разнообразных прикладных задач, требующих использования высокопроизводительных вычислений. Это разнообразие в настоящее время определяет множество стилей и способов написания программ.

В настоящее время практически решены проблемы, связанные с переносимостью программ, написанных для последовательных ЭВМ, построенных на основе принципов, определенных в работах Дж. Фон Неймана (фон-неймановских архитектур). Они преодолеваются различными путями, например:

– созданием языков высокого уровня (ЯВУ), позволяющих писать программы, транслируемые в машинный код (данный подход используется в наиболее распространенных системах программирования);

– разработкой систем, обеспечивающих интерпретацию промежуточного представления, полученного предварительной трансляцией программ, написанных на ЯВУ;

– разработкой интерпретирующих систем, осуществляющих непосредственное выполнение программ, написанных на ЯВУ (возможно, с внутренним преобразованием их в промежуточный код непосредственно перед выполнением);

– динамической генерацией кода объектной машины непосредственно перед исполнением, например, при использовании JIT-компиляции (just in time).

2. Использование последовательных компьютеров

Вычислительные машины, построенные на основе фон-неймановской модели, продолжают широко использоваться в различных предметных областях.

Это объясняется тем, что не везде требуется радикальное повешение производительности. Помимо этого, для систем данного типа накоплен огромный запас программного обеспечения, легко переносимый с одной архитектуры на другую для разнообразных языков программирования и операционных систем.

Подобная переносимость обуславливается тем, что за многие годы разработаны разнообразные трансляторы для различных языков программирования, а высокоуровневая виртуальная последовательная машина, лежащая в основе этих языков, не связана с системой команд конкретного процессора.

Последовательный процессор обрабатывает машинные команды одну за другой. Каждая из команд может использовать данные, хранимые в общей памяти, и полученные в результате выполнения предыдущих команд.

Вместе с тем, максимальная производительность однопроцессорных систем во многом определяется тактовой частотой процессора, которая в настоящий момент достигла определенного предела. Повышение частоты сдерживается за счет повышения выделяемой при работе мощности, что может привести к перегреву.

Поэтому, в настоящее время перспективным направлением считается переход к реализации на одном кристалле нескольких процессоров (ядер).

3. Специфика параллельных вычислений

Главной отличительной особенностью параллельных вычислительных систем является их производительность, то есть, количество операций, производимых за единицу времени.

Различают пиковую и реальную производительность, которая напрямую связана с числом процессоров.

Под пиковой производительностью понимают величину, равную произведению пиковой производительности одного процессора на число таких процессоров в данной машине. Предполагается, что все устройства компьютера работают в максимально производительном режиме. Она является теоретической и недостижимой при запуске приложений.

Реальная производительность, зависит от взаимодействия приложения с архитектурными особенностями используемой системы.

Для оценки эффективности работы вычислительной системы на реальных задачах разработан фиксированный набор тестов. Наиболее известным из них является LINPACK.

Эта программа, предназначена для решения системы линейных алгебраических уравнений с плотной матрицей с выбором главного элемента по строке. LINPACK используется для формирования списка Top500 – пятисот самых мощных компьютеров мира.

В настоящее время большое распространение получили тестовые программы, взятые из разных предметных областей и представляющие собой либо модельные, либо реальные промышленные приложения.

Такие тесты позволяют оценить производительность компьютера действительно на реальных задачах и получить наиболее полное представление об эффективности работы компьютера с конкретным приложением.

Сфера применения *многопроцессорных вычислительных систем* (МВС) непрерывно расширяется, охватывая все новые области в самых различных отраслях науки, бизнеса и производства. Наряду с расширением области применения, по мере совершенствования МВС происходит усложнение и увеличение количества задач в областях, традиционно использующих высокопроизводительную вычислительную технику.

В настоящее время выделен круг фундаментальных и прикладных проблем, эффективное решение которых возможно только с использованием сверхмощной вычислительных ресурсов. Он включает следующие задачи:

– предсказание погоды, климата и глобальных изменений в атмосфере;
– наука о материалах;
– построение полупроводниковых приборов;
– сверхпроводимость;
– структурная биология;
– разработка фармацевтических препаратов;
– генетика;
– квантовая хромодинамика;
– астрономия;
– транспортные задачи;
– гидро- и газодинамика;
– управляемый термоядерный синтез;
– эффективность систем сгорания топлива;
– геоинформационные системы;
– разведка недр;
– наука о мировом океане;
– распознавание и синтез речи;
– распознавание изображений.

2.2. Тема 7: Вычислительные системы с распределенной памятью. Кластерная архитектура. Программное обеспечение систем с распределенной памятью (6 часов).

2.2.1. Перечень и краткое содержание рассматриваемых вопросов:

1. Вычислительные системы с распределенной памятью

MPP архитектуры. Вычислительные систем с распределенной памятью или MPP (massive parallel processing – массивно параллельная обработка) архитектуры в качестве главной особенности отличает физическое разделение памяти по разделам.

Система строится из отдельных модулей, каждый из которых содержит процессор, локальный банк оперативной памяти (ОП), коммутаторы или сетевой адаптер, жесткие диски и/или другие устройства ввода/вывода (рис. 1).

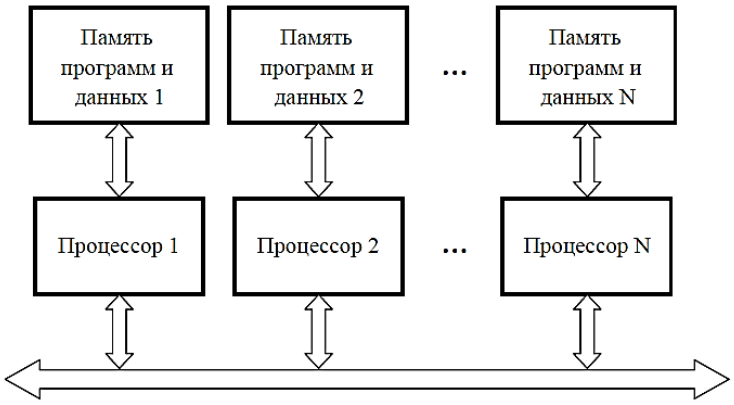

Рис. 1

Один коммутатор используется для передачи команд, другой обеспечивает обмен данными. По сути, модули являются полнофункциональными компьютерами. Доступ к ОП модуля имеют только процессоры из этого же модуля. Модули соединяются специальными коммуникационными каналами. Пользователь может определить логический номер процессора, к которому он подключен, и организовать обмен сообщениями с другими процессорами.

Используются два варианта работы операционной системы (ОС) на машинах MPP архитектуры.

В одном полноценная операционная система (ОС) работает только на управляющей машине, на каждом отдельном модуле работает сильно урезанный вариант ОС, обеспечивающий работу только расположенной в нем ветви параллельного приложения.

Во втором варианте на каждом модуле работает полноценная ОС.

Основным достоинством систем с распределенной памятью является хорошая масштабируемость. Так как каждый процессор имеет доступ только к своей локальной памяти, то нет необходимости в потактовой синхронизации процессоров.

2. Кластерная архитектура

Кластер является одной из разновидностей MPP архитектур.

Он состоит из нескольких компьютеров (часто называемых узлами), объединенных при помощи стандартных сетевых технологий. В качестве узлов могут использоваться серверы, рабочие станции и даже обычные персональные компьютеры.

К достоинствам кластеров можно отнести то, что в случае сбоя одного из узлов другой узел может взять на себя его нагрузку.

Возможности масштабируемости кластеров многократно увеличивают производительность системы.

Кластерные системы являются достаточно дешевыми, поскольку собираются из стандартных комплектующих.

Кластеризация может быть осуществлена на разных уровнях компьютерной системы, включая аппаратное обеспечение, операционные системы, программы-утилиты, системы управления и приложения. Чем больше уровней системы объединены кластерной технологией, тем выше надежность, масштабируемость и управляемость кластера.

Существуют различные варианты построения кластеров. Их можно создать полностью из стандартных комплектующих, что обеспечивает получение дешевых систем, пригодных для экспериментов, учебных целей и отладки параллельных программ, выполнение которых впоследствии планируется на более мощных системах.

В учебных заведениях достаточно часто подобные системы создаются на базе компьютерных классов, включая до нескольких десятков персональных компьютеров.

Кластерная система может быть построена из специально подобранных комплектующих. Этим можно достичь высокой производительности, но при более высокой стоимости. В настоящее время подобные системы получили широкое распространение. Они занимают ведущие позиции в списке Top-500.

3. Программное обеспечение систем с распределенной памятью

MPI (Message Passing Interface, интерфейс передачи сообщений) – наиболее распространенная технология программирования для параллельных компьютеров с распределенной памятью. Реализована в виде библиотеки функций. Поддерживается работа с языками Фортран и Си. Технология основана на обмене сообщениями.

MPI-программа – это множество параллельных взаимодействующих процессов-программ, написанных на каком-либо языке высокого уровня. Каждый процесс работает в своем адресном пространстве, никаких общих переменных или данных в MPI нет. Основным способом взаимодействия между процессами является явная посылка сообщений.

Сообщение – это набор данных некоторого типа. Каждое сообщение имеет несколько атрибутов, в частности, номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения и другие.

Локализация взаимодействия параллельных процессов осуществляется созданием групп процессов со своей средой взаимодействия (коммуникатором) для каждой группы. Группы могут полностью совпадать, входить одна в другую, не пересекаться или пересекаться частично. Процессы могут взаимодействовать только внутри некоторого коммуникатора, сообщения, отправленные в разных коммуникаторах, не пересекаются и не мешают друг другу.

2.3. Тема 8: Вычислительные системы с общей памятью. Программирование для вычислительных систем с общей памятью (8 часов).

2.3.1. Перечень и краткое содержание рассматриваемых вопросов:

1. Вычислительные системы с общей памятью

SMP архитектура (symmetric multiprocessing) – симметричная многопроцессорная архитектура.

Главной особенностью систем с архитектурой SMP является наличие общей физической памяти, разделяемой всеми процессорами (рис. 1).

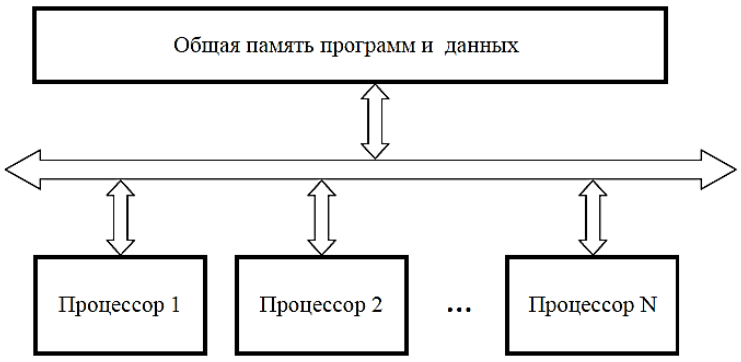

Рис. 1. Схематический вид SMP-архитектуры

Память является способом передачи сообщений между процессорами, при этом все вычислительные устройства при обращении к ней имеют равные права и одну и ту же адресацию для всех ячеек памяти. Поэтому SMP архитектура называется симметричной. Последнее обстоятельство позволяет очень эффективно обмениваться данными с другими

вычислительными устройствами.

SMP-система строится на основе высокоскоростной системной шины, к которой подключаются функциональные блоки трех типов: процессоры (ЦП), операционная система (ОП) и подсистема ввода-вывода (I/O). Для подсоединения к модулям ввода-вывода используются уже более медленные шины каналы.

Вся система работает под управлением единой операционной системы, которая автоматически (в процессе работы) распределяет процессы по процессорам. Однако иногда возможна и явная привязка.

Основные преимущества SMP-систем.

1) Простота и универсальность для программирования.

Архитектура SMP не накладывает ограничений на модель программирования, используемую при создании приложения: обычно используется модель параллельных ветвей, когда все процессоры работают абсолютно независимо друг от друга. Однако, можно реализовать и модели, использующие межпроцессорный обмен.

Использование общей памяти увеличивает скорость такого обмена, пользователь также имеет доступ сразу ко всему объему памяти.

Для SMP-систем существуют сравнительно эффективные средства автоматического распараллеливания.

2) Легкость в эксплуатации.

Как правило, SMP-системы используют систему охлаждения, основанную на воздушном кондиционировании, что облегчает их техническое обслуживание.

3) Относительно невысокая цена.

Недостаток – системы с общей памятью, построенные на системной шине, плохо масштабируемы.

2 Программирование для вычислительных систем с общей памятью

Основной моделью программирования, эффективно использующей вычислительный потенциал SMP архитектур, является совокупность параллельных потоков управления, каждый из которых запускается на отдельном процессоре и совместно с другими потоками использует общую память в рамках единого процесса. Такие программы называются *многопоточными*.

Все нити одного процесса совместно используют его структуры и данные, такие как дескрипторы файлов, идентификаторы пользователей. Следовательно, потоки имеют доступ к одним и тем же функциям, данным, открытым файлам и всему остальному.

Многопоточная программа запускается с одного главного потока, которая затем может создавать новые потоки путем вызова соответствующих подпрограмм. Новый поток сразу же запускает подпрограмму, обеспечивая, тем самым, создание других команд, оперирующих с данными в том же пространстве адресов.

Когда много потоков используют одни и те же данные, они координируют свою работу посредством синхронизирующих переменных, таких как примитивы взаимного исключения (мьютексы).

Другим способом синхронизации потоков является их взаимодействие путем непосредственного обмена сигналами.

2.4. Тема 9: Системы на основе программируемых логических интегральных схем. Параллельные вычислительные системы на основе ПЛИС. Программирование вычислительных систем на ПЛИС (4 часов).

2.4.1. Перечень и краткое содержание рассматриваемых вопросов:

1. Параллельные вычислительные системы на основе ПЛИС

Программируемые логические интегральные схемы (ПЛИС) или микросхемы

программируемой логики – одна из привлекательных технологий, предоставляющая возможности быстрого создания цифровых систем с произвольной внутренней структурой.

Разработка устройств на ПЛИС занимает значительно меньше времени по сравнению с разработкой специализированных микросхем. Она также гораздо дешевле за счет того, что изменение принципиальной схемы выполняется перепрограммированием одного и того же экземпляра микросхемы.

На основе ПЛИС разрабатываются реконфигурируемые многопроцессорные вычислительные системы. В отличие от МВС с «жесткой» архитектурой, в частности, кластерных суперЭВМ, архитектура реконфигурируемых систем может изменяться в процессе ее функционирования. В результате у пользователя появляется возможность адаптации архитектуры системы под структуру решаемой задачи или, иными словами, создания в рамках универсальной среды проблемно-ориентированных многопроцессорных вычислительных структур. Реализация данной концепции обеспечивает высокую реальную производительность многопроцессорной вычислительной системы, близкую к пиковой, на широком классе задач, а также практически линейный рост про-изводительности при увеличении числа процессоров в системе.

Реализация и широкое внедрение в практику идеи реконфигурируемых многопроцессорных вычислительных систем стало возможным с появлением ПЛИС со сверхвысокой степенью интеграции, на базе которых стало возможным создание больших вычисли-тельных полей. Это позволило в полной мере реализовать преимущества реконфигурируемых мультиконвейерных вычислительных структур, существенно опережающих многопроцессорные системы с традиционной «жесткой» архитектурой по таким характеристикам как соотношения «реальная производительность/пиковая производительность» (эффективность) и «реальная производительность/объем» (компактность).

Следует отметить, что по этому пути уже идут и многие ведущие миро-вые производители. Компания Cray предлагает снабдить процессоры Opteron на двухпроцессорной платформе дополнительным модулем на базе ПЛИС. На этот модуль возлагается решение трудоемких для универсального процессора задач.

Различными фирмами разработаны реконфигурируемые устройства в виде плат сопроцессоров для задач цифровой обработки сигналов и моделирования специализированных устройств.

Компанией Stretch предлагается реконфигурируемый процессор, представляющий собой сочетание известной микро-процессорной архитектуры RISC с кристаллами ПЛИС. Разработанный компилятор автоматически выделяет в программе места, требующие интенсивных вычислений, и реализует их с помощью ПЛИС.

Компания Star Bridge Systems одна из первых представила суперкомпьютер, который основан на реконфигурируемой технологии и построен на кристаллах ПЛИС фирмы Xilinx.

В исследовательском центре Berkeley Wireless Research Center также разработана суперкомпьютерная система High-End Reconfigurable Computing System на кристаллах ПЛИС.

По мнению разработчиков, использование кристаллов ПЛИС обеспечивает более чем на два порядка большую производительность систем в сравнении с производительностями систем с аналогичной потребляемой мощностью и стоимостью, реализованными на стандартных микропроцессорах.

2. Программирование вычислительных систем на ПЛИС

Программирование систем, реализованных на ПЛИС по сути означает формирование новой топологии интегральной схемы.

Одним из низкоуровневых языков, широко используемых при проектировании топологии сверхбольших интегральных схем (СБИС) на ПЛИС, является VHDL (Very high speed integrated circuits Hardware Description Language). Он является базовым языком при разработке аппаратуры современных вычислительных систем. Первоначально язык предназначался для моделирования, но позднее из него было выделено синтезируемое подмножество.

Написание модели на синтезируемом подмножестве позволяет автоматический синтез схемы функционально эквивалентной исходной модели. Средствами языка VHDL возможно проектирование на различных уровнях абстракции (поведенческом или алгоритмическом, регистровых передач, структурном), в соответствии с техническим заданием и предпочтениями разработчика. Заложена возможность иерархического проектирования, максимально реализующая себя в экстремально больших проектах с участием большой группы разработчиков.

Представляется возможным выделить следующие три составные части языка:

– алгоритмическую, основанную на языках Ada и Pascal и придающую языку VHDL свойства языков программирования;

– проблемно ориентированную – в сущности и обращающую VHDL в язык описания аппаратуры;

– объектно-ориентированную, интенсивно развиваемую в последнее время.

2.5. Тема 10: Системы на основе графических процессоров. Программирование графических процессоров. Программирование с использованием CUDA (10 часов).

2.5.1. Перечень и краткое содержание рассматриваемых вопросов:

1. Системы на основе графических процессоров (GPU)

В настоящее время широкое применение в высокопроизводительных вычислениях находят графические процессоры (Graphics Processing Unit, GPU). Отличительными особенностями GPU по сравнению с традиционными и многоядерными процессорами являются:

– архитектура, максимально нацеленная на увеличение скорости расчѐта текстур и сложных графических объектов;

– ограниченный набор команд.

Зародившись изначально как специализированные устройства для выполнения эффективной обработки и отображения компьютерной графики благодаря специализированной конвейерной архитектуре, графические процессоры впоследствии превратились в мощные вычислители. Оказалось, что многие ресурсоемкие графические задачи хорошо ложатся на архитектуру GPU. В результате возникло направление их использования, связанное с решением неграфических задач (General-Purpose Computing on Graphics Processing Unit, GPGPU).

GPU изначально реализованы как параллельные системы, содержащие множество процессорных ядер, взаимодействующих через свою общую память. Использование этой памяти позволяет сократить число обращений к оперативному запоминающему устройству центрального процессора, с которым графический процессор обычно используется совместно. Обычно центральный процессор обеспечивает подготовку данных для вычислений и взаимодействие с устройствами ввода-вывода.

Общая схема графического процессора приведена на рис. 1.

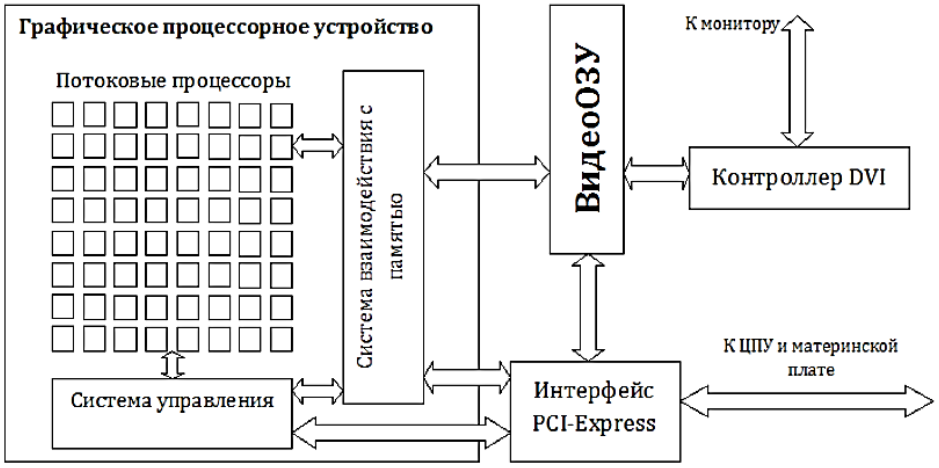

Рис. 1. Обобщенная структура графического процессора

Изначально графические ускорители разрабатывались с использованием архитектуры SIMD (Single Instruction, Multiple Data), при которой одна машинная команда одновременно выполнялась над множеством данных. Это было обусловлено практическим отсутствием взаимозависимостей между отдельными пикселями обрабатываемых графических изображений. Однако ориентация на более общие вычислительные задачи изменили организацию системы, которая стала одновременно выполнять множество *нитей*, каждая из которых инициируется одной и той же инструкцией.

Специфической особенностью этих архитектур, построенных по принципу SIMT (Single Instruction, Multiple Thread), является то, что нити запускаются и выполняются не одновременно. Они объединяются в более мелкие группы, размер которых зависит от физических ресурсов графического ускорителя. Обычно для решения задачи используется большое количество нитей, каждой из которых соответствует один элемент обрабатываемых данных.

2. Программирование графических процессоров

Существуют различные системы программирования для графических процессоров. Все они ориентированы на то, что программа пишется как для центрального процессорного узла, так и для GPU.

Однако, если для центрального процессора достаточно использовать традиционные языки, то графический процессор, имеющий специфическую архитектуру, программируется на специализированных языках, таких как Cg, GLSL, HLSL. Часть программы, написанная на традиционном языке, отвечает за подготовку и передачу данных, а также за запуск GPU-программы (шейдера).

Вместе с тем, такой подход обладает определенными недостатками, связанными с тем, что использование возможностей графического процессора происходит через программный интерфейс (API) библиотек, ориентированных на работу с графикой, таких как OpenGL или Direct3D. В них отсутствует возможность организации взаимной обработки связанных данных (которые в библиотеках представляют отдельные пиксели), а также ряд других возможностей, широко используемых в параллельных программах.

Поэтому появились специальные инструментальные средства, ориентированные на решение высоко-производительных задач на GPU.

3. Программирование с использованием CUDA

Технология CUDA (Compute Unified Device Architecture) предложена компанией Nvidia для программирования ее графических процессоров.

Она заметно облегчает разработку GPGPU-приложений. Все программы пишутся на языке, являющемся расширением языка программирования C. Технология поддерживается множеством разработанных свободных библиотек, хорошей документацией и является кроссплатформенной.

CUDA основывается на подходе, при котором GPU рассматривается как MPP сопроцессор, который является сопроцессором к центральному процессору (ЦП), обладает собственной памятью, может одновременно выполнять огромное количество отдельных нитей.

В отличие от нитей ЦП:

– нити GPU имеют крайне низкую стоимость создания, управления и уничтожения;

– для эффективной загрузки GPU необходимо использовать много тысяч нитей (обычный центральный процессор перегружается уже при нескольких десятках нитей).

Специфика организации вычислений ведет к тому, что нити образуют специальную иерархию (рис. 2).

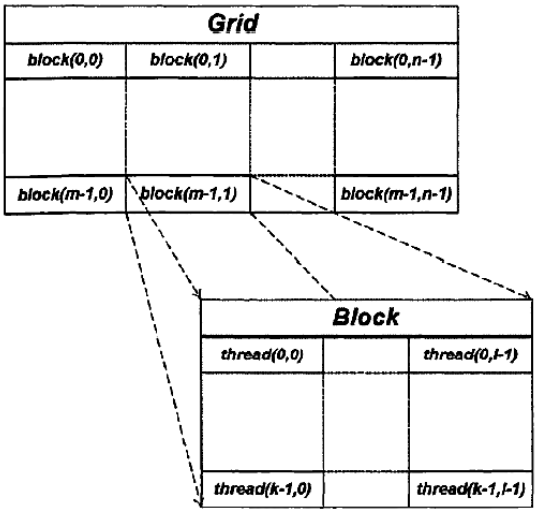

Рис. 2. Иерархия нитей в CUDA

Верхний уровень иерархии – сетка (grid) соответствует всем выполняемым нитям. Он представляет одно- или двухмерный массив блоков (block). Каждый блок в свою очередь является одно-, двух- или трехмерным массивом нитей (thread). Каждый блок имеет свой адрес, задаваемый индексами в сетке, а каждая из нитей индексируется внутри соответствующего блока.

2.6. Тема 11: Гибридные параллельные архитектуры. Проблемы переносимости параллельных программ. Распараллеливание последовательных программ (4 часа).

2.6.1. Перечень и краткое содержание рассматриваемых вопросов:

1. Гибридные параллельные архитектуры

Гибридные архитектуры формируются путем объединения в единую вычислительную систему разнотипных компьютеров. В частности, появление многоядерных компьютеров привело к тому, что большие кластерные системы стали состоять из узлов, по сути являющихся SMP-системами. В настоящее время интенсивно развиваются гибридные кластерные системы, которые наряду со ставшими обычными SMP-узлами содержат графические ускорители. Это позволило значительно поднять пиковую производительность современных компьютеров. В списках Top-500 последних

редакции эти гибриды прочно обосновались в первой десятке.

Переход к гибридным архитектурам еще больше усложнил программирование параллельных вычислительных систем, так как, для повышения эффективности программы, появилась необходимость учитывать использование многопоточных и много процессных вычислений, специфику графических процессоров а также работу с общей памятью разного типа. Постоянная смена архитектур все больше и больше требует переписывания ранее написанных программ в угоду повышения их вычислительных возможностей. Это, в свою очередь, поднимает вопрос о поиске новых путей написания параллельных программ, обеспечивающих их большую гибкость к постоянно изменяющимся условиями эксплуатации.

2. Проблемы переносимости параллельных программ

Разработка параллельных программ в основном обуславливается используемыми методами алгоритмизации и управления параллельными вычислениями. Это объясняется тем, что основной проблемой является необходимость организации их эффективного выполнения на конкретной системе и при решении конкретной задачи. Определяющую роль в написании параллельных программ играет выбор архитектуры исполнителя, а не то, как будет иерархически организована программа, и по какой технологии она будет выполняться. Это ведет к ситуации, при которой разрабатываемые параллельные программы становятся практически непереносимыми с одной архитектуры на другую.

Существующие методы разработки архитектурно независимых параллельных программ, в свою очередь не обеспечивают их эффективного выполнения.

В настоящее время практически решены проблемы переноса последовательных программ с одной вычислительной системы на другую. Существуют разнообразные по стилю языки программирования, обеспечивающие написание программ, выполняемых на различных архитектурах и под управлением разнообразных операционных систем. При этом парадигма программирования не играет существенной роли. Примеры использования одного и того же языка на различных платформах можно привести как для отдельных стилей, так и для инструментальных систем, обеспечивающих поддержку мультипарадигма-тических подходов. Перенос осуществляется независимо от того, каким путем осуществляется преобразование исходной программы в программу конкретного исполнителя.

Иначе обстоит дело с параллельным программированием. В этом случае невозможно предложить единый универсальный исполнитель, обеспечивающий одинаково эффективный способ написания параллельных программ. Проблемы, возникающие с переходом к параллельному программированию, обуславливаются разнообразием факторов, влияющих на организацию эффективных вычислений. К стилю написания программы добавляется учет числа используемых ресурсов и их вычислительной мощности. Резкое изменение производительности возможно даже при изменении параметров одного из ресурсов участвующих в вычислениях (например, при изменении пропускной способности коммутатора). Все это ведет к тому, что для организации эффективных вычислений необходимо каждый раз перестраивать структуру программы или использовать внутри нее дополнительные модули настройки на исполнительную среду.

Попытки создания инструментальных средств, обеспечивающих переносимость параллельных программ, предпринимались неоднократно и осуществлялись по нескольким направлениям. Все они, тем или иным образом, связаны с написанием программ, для некоторой обобщенной архитектуры. Выполнение подобных программ на ПВС осуществляется после их предварительного преобразования с использованием преобразователей параллелизма.

Существуют различные варианты классификации методов преобразования параллельных программ. Одним из них является деление по способам представления

исходного параллелизма:

1) Использование последовательного программирования с дальнейшим автоматическим распараллеливанием разработанных программ.

2) Применение языков программирования, обеспечивающих непосредственное описание любых потоков параллельного управления, по сути являющихся расширениями традиционных последовательных языков за счет добавления разнообразных примитивов распараллеливания и синхронизации. При этом уровень параллелизма полностью контролируется разработчиком и может задаваться в зависимости от конкретных условий. Преобразование подобных программ является переводом одной параллельной формы в другую.

3) Опора на языки, неявным образом описывающие неограниченный параллелизм, что обеспечивается заданием только информационных связей. Предполагается, что программа будет рассчитана на выполнение в виртуальной машине с неограниченными ресурсами, а запуск операторов будет происходить по готовности их данных. Адаптация программы к конкретной ПВС обычно осуществляется путем сжатия параллелизма с учетом ресурсных и архитектурных ограничений.

Каждый из подходов в настоящее время представлен семейством соответствующих инструментальных средств, имеет достоинства и недостатки.

3. Распараллеливание последовательных программ

Последовательное программирование выглядит весьма привлекательным, так как, на первый взгляд, позволяет:

- применять опыт, накопленный в течение многолетней эксплуатации вычислительных машин с традиционной архитектурой;
- разрабатывать последовательные программы, не зависящие от особенностей параллельной вычислительной системы;
- использовать продукты, накопленные за многие годы создания программного обеспечения.
- не заботиться о параллельной организации программы, полагаясь в дальнейшем на то, что система автоматического распараллеливания решит все вопросы по эффективному переносу программ на любую параллельную систему.

В свое время этот подход достаточно широко и разносторонне исследовался. Однако многие из возлагаемых надежд не оправдались. В настоящее время он практически не используется в промышленном программировании, так как обладает следующими недостатками:

- распараллеливание последовательных программ очень редко обеспечивает достижение приемлемого уровня параллелизма из-за ограничений вычислительного метода, выбранного для построения соответствующей программы, что часто обуславливается стереотипами последовательного мышления;
- в реальной ситуации учесть особенности различных параллельных систем оказалось гораздо труднее, чем это изначально предполагалось.

2.7. Тема 12: Явное написание параллельных программ и их преобразование. Использование функциональных и потоковых языков (6 часов).

2.7.1. Перечень и краткое содержание рассматриваемых вопросов:

1. Явное написание параллельных программ и их преобразование

Практически во все времена наиболее популярным был подход, опирающийся на языки программирования, учитывающие особенности архитектуры системы. Несомненным его достоинством является максимальное использование возможностей предоставляемых техническими средствами. В частности, программист, опираясь на

знание особенностей параллельной вычислительной системы, может весьма эффективно распределять задачу по конкретным вычислительным ресурсам. Явно указывая потоки управления, он ограничивает или расширяет параллелизм задачи с учетом пропускной способности и производительности различных узлов ПВС.

Следует также отметить, что часть задач носит предметно ориентированный характер. Это ведет к определенным удобствам при их реализации на ряде ПВС. Каждому процессу ставится в соответствие некоторый вычислительный ресурс, а взаимодействие процессов описывается на уровне взаимодействия ресурсов. Ресурсы могут создаваться и уничтожаться в ходе вычислений, связываться между собой как по информационным, так и по управляющим каналам. К таким задачам можно отнести:

- имитационное моделирование систем, рассматривающее ограниченные ресурсы как взаимодействующие процессы, непосредственно и постоянно взаимодействующие между собой;

- управление в реальном масштабе времени, когда взаимодействие осуществляется между реальными, одновременно функционирующими параллельными ресурсами.

В этом случае гораздо удобнее использовать языки программирования, поддерживающие описание решаемой задачи за счет явного задания потоков управления, объектного программирования и непосредственного управления ресурсами.

Параллельное программирование для конкретных архитектур широко использовалось в конце 70-х, начале 80-х годов. В это время были разработаны различные вычислительные системы, значительно отличающиеся друг от друга. В дальнейшем развитие технологии создания однокристальных микропроцессоров привело к резкому снижению эффективности подобных разработок и к нецелесообразности создания нетрадиционных архитектур. Это привело к использованию при создании высокопроизводительных систем стандартных аппаратных средств: симметричных мультипроцессоров, кластеров, массово-параллельных систем.

Использование подобных архитектур позволило унифицировать методы разработки параллельных программ, увязав их не с конкретными вычислительными системами, а с семействами архитектур ПВС. Появились легко переносимые инструментальные средства и пакеты, позволяющие выполнять одни и те же программы на вычислительных системах, имеющих разные системы команд.

2. Использование функциональных и потоковых языков

Создание прикладных параллельных программ, ориентированных на численные вычисления и символьную обработку, удобнее осуществлять с применением функциональных языков параллельного программирования, в которых выполнение каждого оператора осуществляется по готовности его данных. Они не требуют явного описания параллелизма задачи. Достаточно указать информационную взаимосвязь между функциями, осуществляющими преобразование данных. Использование таких языков позволяет:

- создавать программы с параллелизмом на уровне операторов, ограниченным лишь методом решения задачи;

- обеспечивать перенос программы на конкретную архитектуру, не распараллеливая программу, а сжимая ее максимальный параллелизм (перенос подобных программ никоим образом не связан попытками преобразования структур управления и построением информационного графа, так как данный граф изначально построен);

- проводить оптимизацию программы по множеству параметров с учетом специфики архитектуры ВС, для которой осуществляется трансляция без учета управляющих связей программы.

Существуют различные модели параллельных вычислений и языки, построенные на их основе, которые, для описания параллелизма, используют только информационные зависимости между выполняемыми функциями. Работы в этой области условно можно

разделить на два направления:

- разработка методов управления вычислениями по готовности данных;
- разработка языков и методов функционального программирования.

2.8. Тема 13: Специфика управления вычислениями. Перспективы архитектурно независимого параллельного программирования (4 часа).

2.8.1. Перечень и краткое содержание рассматриваемых вопросов:

1. Специфика управления вычислениями

Для определения специфики модели вычислений (МВ), используемой в языке, обеспечивающем создание переносимых параллельных программ необходимо проанализировать факторы, определяющие особенности вычислительных процессов. Анализ базируется на общности информационного графа решаемой задачи и отличиях, возникающих в исполнении задаваемых им операций на различных вычислительных системах. Эти отличия определяются составом вычислительных ресурсов, методами управления ими, особенностью управления вычислениями в самой программе.

Для выбора МВ, необходимо построить классификацию возможных стратегий управления параллельными вычислениями, учитывающую следующие факторы:

- отношения между операциями, составляющими алгоритм решаемой задачи, и обрабатываемыми ими данными;
- зависимость от вычислительных ресурсов;
- зависимость от субъективных управляющих воздействий.

Учет отношений между данными и операциями осуществляется исходя из того, что любая программа разрабатывается для выполнения конкретных вычислений. При этом зависимость момента выполнения операций от наличия аргументов определяется информационным графом решаемой задачи. Корректное выполнение любой операции обработки данных возможно только в том случае, если, в ходе предварительных вычислений, будут сформированы значения всех ее операндов.

Зависимость от ресурсов вычислительных систем связана с тем, что они не являются безграничными. Поэтому, даже в том случае, если программа может использовать бесконечное число вычислительных ресурсов, на нее приходится накладывать ограничения, обуславливаемые архитектурой вычислительной системы. Эти ограничения снижают параллелизм задачи и требуют ввода дополнительных механизмов управления вычислениями.

Зависимость от субъективных управляющих воздействий показывает, что выполнение отдельных операций определяется моментом их запуска, который выбирается не только по объективным причинам (готовность аргументов и наличие вычислительных ресурсов). Зачастую программист руководствуется внутренним восприятием, обуславливаемым его пониманием разрабатываемого алгоритма.

Наличие управляющих воздействий, определяющих граф потоков управления, является неотъемлемой частью алгоритма решаемой задачи.

2. Перспективы архитектурно независимого параллельного программирования

Несмотря на определенные достижения в области создания программ на основе неявного управления вычислениями, проблемы, связанные с построением параллельных программ, свободных от всех ресурсных ограничений, до конца остаются неразрешенными. Это обуславливается наличием соответствующих ограничений, как в моделях вычислений, так и в языках программирования. Для преодоления ресурсных ограничений, необходимо предложить методы, разработанные на основе модели вычислений, использующей концепцию неограниченных ресурсов и неявное управление по готовности данных.

Рассмотренные выше функционально-потоковые языки в основном ориентированы на эффективное решение конкретных прикладных задач, что ведет к использованию смешанных моделей вычислений, обеспечивающих явное управление ресурсами. Практическое отсутствие интереса к созданию «идеальных» систем, обеспечивающих разработку переносимых параллельных программ, во многом обуславливается семантическим разрывом между методами написания программ, которые должны использовать параллелизм на уровне операций, и современными техническими средствами, обеспечивающими реальную поддержку только крупноблочного распараллеливания. Эффективнее и проще осуществлять кодирование, непосредственно поддерживающее архитектурно зависимые методы.

Вместе с тем, исследования в области создания мобильных, ресурсно-независимых параллельных программ представляют интерес, так как, в перспективе позволяют избежать переписывания кода при изменении архитектур вычислительных систем. Усилия должны быть направлены не на создание языка, являющегося основой машинной архитектуры, а на решение следующих задач:

- исследование и разработку систем программирования, обеспечивающих написание программ, не зависящих от архитектур конкретных ВС;
- разработка методов преобразования параллелизма архитектурно независимых программ в архитектурно зависимые параллельные программы;
- разработку промежуточных интерпретирующих средств, позволяющих выполнять архитектурно независимые программы на реальных системах.

Это позволит приступить к накоплению программного обеспечения, которое не придется переписывать заново при неизбежном изменении вычислительных систем. Достаточно будет разработать новые методы преобразования, предназначенные для «сжатия» максимального параллелизма.

2.9. Тема 14: Модель вычислительного процесса. Стратегии управления в вычислительных системах (4 часа).

2.9.1. Перечень и краткое содержание рассматриваемых вопросов:

1. Модель вычислительного процесса

Параллельная вычислительная система – это инструмент, обеспечивающей решение целевой задачи.

Выполнение программы определяет процесс (P) обработки данных, разбиваемый на отдельные действия, каждое из которых осуществляет обработку пакета (I), состоящего из операции и обрабатываемых ею данных (операционного пакета). Сведения об операции берутся из написанной программы. Данные формируются в ходе вычислений.

Связь между процессами и ресурсами вычислительной системы

Определим вычисления, задающие процесс P, протекающий в вычислительных ресурсах R, как обработку операционного пакета I_{in}, в результате чего формируется выходной пакет I_{out}. Пусть момент начала вычислений определяется входным управляющим воздействием C_{in}, а их завершение фиксируется по выдаче вычислительным ресурсом управляющего сигнала C_{out}.

Графическая интерпретация этих вычислений представлена на рис. 1.

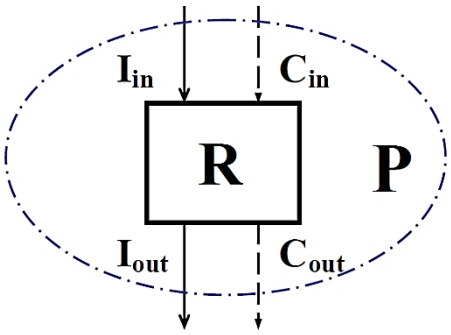

Рис. 1. Процесс как вычисления, протекающие в ресурсе

Иерархическая организация вычислений позволяет говорить о том, что процесс можно разделить на более мелкие подпроцессы, каждый из которых может выполняться в отведенной для этого части общих ресурсов. При этом, одни и те же ресурсы могут повторно использоваться при выполнении различных вычислений, разделяясь подпроцессами во времени.

Процессы протекают в ресурсах вычислительной системы (ВС), обобщенная структура которой приведена на рис. 2.

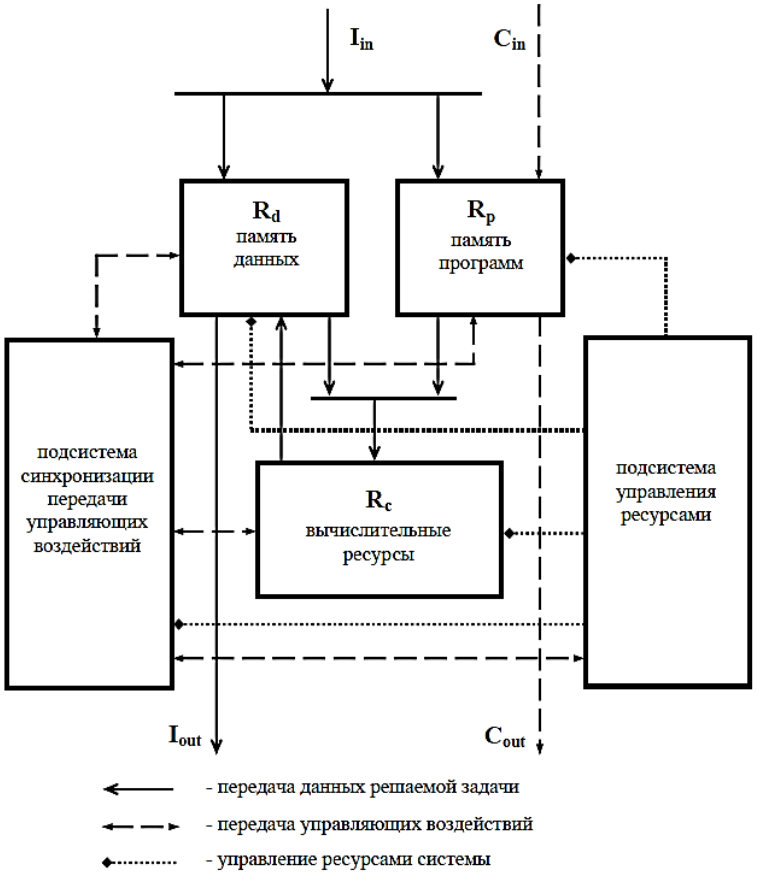

Рис. 2. Потоки, определяющие протекание процессов в вычислительной системе

2. Стратегии управления в вычислительных системах

Вычислительный процесс, определяемый некоторым алгоритмом, можно рассматривать как совокупность элементарных процессов, каждый из которых обеспечивает выполнение простой операции. Для корректного выполнения необходимо, чтобы к моменту запуска элементарного процесса соблюдались определенные условия.

Условие готовности данных (Information – условие). Перед запуском процесса на его информационных входах должны присутствовать все необходимые аргументы и функции, составляющие операционный пакет. Отсутствие каких-либо данных к моменту запуска процесса ведет к получению неправильного результата. То есть, если существуют процессы, результаты которых являются аргументами рассматриваемого процесса, то они должны завершиться к моменту запуска текущего процесса.

Условие выделения ресурсов (Resource – условие). Выполнение процесса протекает в соответствующих ресурсах, поэтому, перед его запуском, эти ресурсы должны быть определены и предоставлены. Ресурсный вход должен содержать информацию, подтверждающую выделение ресурсов, необходимых для успешного выполнения процесса. В противном случае его запуск просто невозможен.

Условия подтверждения использования результатов (Acknowledge – условие). Ресурсы, занимаемые процессом, могут освобождаться или повторно использоваться только после того, как результаты вычислений будут использованы всеми процессами, являющихся приемниками этих результатов в качестве аргументов. В противном случае данные будут потеряны, что приведет к неправильным вычислениям. Необходимость задержки в освобождении ресурсов мотивируется тем, что память, используемая для хранения операндов, является разделяемым ресурсом. В нее записываются результаты вычислений элементарного процесса. Из нее же происходит чтение аргументов, необходимых другим процессам.

Информация о выполнении условий готовности в рассматриваемой модели подтверждается наборами соответствующих управляющих сигналов:

– сигналами готовности элементов операционного пакета;

– сигналами готовности необходимых вычислительных ресурсов;

– сигналами, подтверждающими отсутствие конфликтов при использовании разделяемых ресурсов.

Эти входные сигналы формируются на основе выходных сигналов завершающихся процессов. Завершение элементарного вычислительного процесса одновременно подтверждает готовность результатов и использование им, в качестве своих аргументов, результатов, полученных ранее в процессах – «передатчиках». Естественно, что при более сложном представлении процессов, схема их взаимодействия и моменты формирования управляющих сигналов тоже будут сложнее.

Моменты выдачи управляющих сигналов и сами принципы их формирования, определяющие наступление соответствующих событий, могут осуществляться одним из следующих способов:

1) Явно, когда при описании взаимодействия процессов (в ходе написания программы) разработчик сам создает логику порождения и проверки управляющих сигналов, определяет пути и моменты их передачи. Назовем такое формирование управления субъективным (Subjective)

2) Неявно, когда, при описании процессов, механизм управления тем или иным условием готовности просто не задается из расчета, что процессы и так будут выполняться корректно.

2.10. Тема **15:** Стратегии управления в языках программирования. Связь стратегий с переносимостью параллельных программ (6 часов).

2.10.1. Перечень и краткое содержание рассматриваемых вопросов:

1. Стратегии управления в языках программирования

Особенностью языков программирования является то, что неявное управление в них с помощью специальных средств не задается.

Следовательно, при написании программ автоматическое и пустое управление не различаются. Тогда, для трех условий готовности существуют только два механизма управления: субъективный и неявный (Unevident). Поэтому, в языках программирования можно описать только восемь основных стратегий управления вычислениями:

$$|SL| = X^Z,$$

где SL – множество стратегий управления в языках программирования,
X – трехэлементное множество условий готовности,
Z – двухэлементное множество методов управления процессом:
Z = {Subjective, Unevident}.

2. Связь стратегий с переносимостью параллельных программ

Использование стратегий управления для классификации методов программирования позволяет рассмотреть ограничения, накладываемые на переносимость, различными подходами к параллельному программированию.

Специфика управления при последовательном программировании

Все последовательные парадигмы программирования определяют только один поток управления, осуществляющий запуск одного процесса, и, по его завершению, передачу управляющего сигнала следующему процессу. Однако, между ними имеются и небольшие различия.

Императивное программирование базируется на создании программы из отдельных команд, определяемых фон-неймановской архитектурой.

В целом не существует никаких ограничений на порядок их следования в программе, поэтому программист должен использовать субъективную стратегию для организации правильного соответствия между управлением и информационным графом решаемой задачи. При управлении ресурсами используется неявная стратегия для управления подтверждениями, что гарантирует правильность выполнения программы только при обработке одного независимого набора данных. Для выделения процессорного ресурса тоже используется неявное управление, так как отсутствуют проблемы в его предоставлении одному процессу. Субъективное управление памятью, используемой для хранения результатов промежуточных вычислений, определяется базовой архитектурой и может вести к искажению информационного графа, что не позволяет его использовать без программы, задающей последовательность вычислений. По этой же причине императивное программирование подвержено дополнительным ошибкам, связанным с неправильным использованием ячеек памяти.

Для того чтобы по императивной программе восстановить информационный граф, необходим дополнительный анализ ее структуры, связанный с тщательным исследованием зависимости между графом управления и связываемыми им командами.

Дополнительной проблемой, затрудняющей распараллеливание, являются попытки программиста оптимизировать задачу под последовательные вычисления. Это накладывает определенные ограничения на используемые структуры данных и алгоритмы, что в дальнейшем зачастую не позволяет выявить максимальный параллелизм, присущий решаемой задаче.

Автоматное программирование определяет структуру программы практически так же, как и императивное программирование. Основным его отличием является акцент при разработке программу на логику переходов, обуславливаемую концентрацией условий в состояниях. Однако при этом сохраняется последовательный характер переходов, что ведет к использованию вычислителя, практически эквивалентного исполнителю императивных программ. Еще большее сходство наблюдается в реализации функций выхода и анализе условий перехода. При выполнении этих операций обычно используется явная работа с памятью. Практическая эквивалентность императивного и автоматного программирования подтверждается легкостью перехода от одной модели к другой, как в

одном, так и другом направлении, что широко применяется в практическом программировании путем использования различных методик.

Специфика субъективного управления при параллельном программировании

Субъективное управление вычислениями в параллельном программировании используется гораздо шире, чем в последовательном программировании. Это обуславливается как наличием дополнительных ресурсов, так и их синхронизацией, необходимой для достижения корректности вычислений.

На вычислительные ресурсы накладывается гораздо больше ограничений, чем на память, поэтому явное управление ими позволяет в настоящее время получать более высокую производительность выполнения программ по сравнению с использованием алгоритмов динамического или статического управления ресурсами, базирующихся на неявном управлении. Во многом это обуславливается отсутствием эффективных методов преобразования параллельных программ, построенных на основе неявного управления вычислениями в программы для конкретных архитектур.

Вместе с тем следует отметить, что появление подобных алгоритмов связано с накоплением знаний в данной предметной области и может значительно изменить существующее положение, повысив мобильность параллельных программ. Аналогом подобного процесса развития может служить использование памяти при последовательном программировании, что позволило, в настоящее время, создать функциональные языки программирования, для которых эффективность генерируемого кода сопоставима с эффективностью кода императивных высокоуровневых языков.

Раздел 3. Парадигмы программирования и проблемы эволюционного расширения программ

3.1. Тема 16: Сущность идеи эволюционной разработки программ. Языковые средства эволюционной разработки программ (4 часа).

3.1.1. Перечень и краткое содержание рассматриваемых вопросов:

1. Сущность идеи эволюционной разработки программ

Несмотря на подавляющий перевес объектно-ориентированного подхода, современная разработка программ осуществляется с применением различных парадигм программирования.

Во многом это обуславливается тем, что достижения требуемых критериев качества использование только одного стиля бывает недостаточно. Даже используя языки программирования, считающиеся объектно-ориентированными (такие как Java или C#), многие программисты уходят от чистого стиля, разбавляя его процедурными или функциональными вкраплениями.

Причина этого заключается в том, что в каждой из парадигм существуют свои подходы к реализации одних и тех же программных объектов с разной степенью эффективности. Поэтому использование одного подхода ведет к потере эффективности или несоответствию программы требуемым критериям качества.

Постепенное расширение программной системы – один из основных критериев, обуславливающих ее успешное создание и эксплуатацию. Невозможно за один проектный цикл построить большую программу, удовлетворяющую всем предъявляемым требованиям, что объясняется следующими факторами.

1) Требования к программному продукту могут меняться не только во время разработки, но и во время эксплуатации. Выявляется потребность в новых функциях, появляются новые условия использования. Все это ведет к необходимости вносить дополнительные расширения в уже написанный код.

2) Разработка больших программных систем – длительный и кропотливый процесс,

требующий тщательной разработки и отладки. При попытках создать все систему сразу разработчики сталкиваются с проблемами, обуславливаемыми большой размерностью решаемой задачи. Поэтому, даже при хорошо известном наборе реализуемых функций, целесообразно вести инкрементальную разработку программ, постепенно добавляя и отлаживая новые функции на каждом витке итеративного цикла разработки. Сдача программы в эксплуатации при этом осуществляется итеративно.

Эволюционное развитие программы невозможно без использования специальных методов разработки и соответствующих систем программирования. Исследования в этом направлении проводились еще в период расцвета структурного программирования и были направлены на совершенствование восходящего и нисходящего подходов.

Разработка подобных сред, обладающих высокой эффективностью, вначале 1980-х гг. ограничивалась возможностями существующих вычислительных систем. В более поздний период данное направление не получило развития, что во многом обусловлено ростом популярности объектно-ориентированного программирования (ООП).

Использование объектно-ориентированного подхода привело к значительным успехам в области эволюционной разработки программного обеспечения.

Это было связано с возможностями создания новых программных объектов на основе уже существующих и подменой старых объектов новыми без изменения интерфейса между клиентской и обслуживающей частями программы. Технически подобный прием обеспечивался использованием механизмов наследования и виртуализации. Эффективность этого приема позволила использовать его в самых разнообразных ситуациях в качестве универсального концепта и привела к созданию на его основе объектно-ориентированной методологии (ООМ).

В рамках ООМ разработаны и успешно развиваются в настоящее время различные идеи эволюционного расширения программ. В частности:

- методы инкрементального проектирования унифицированы в рамках универсального процесса разработки;
- предложены и экспериментально подтверждены приемы эволюционной разработки программ на различных ОО языках программирования, оформленные в виде образцов (паттернов).

2. Языковые средства эволюционной разработки программ

Инструментальные и языковые средства, в том или ином виде поддерживающие безболезненное расширение программ, стали появляться вместе с первыми языками программирования.

Основная их идея заключается в распределенном формировании программных объектов, определяющих как расширение данных, так и добавление нового кода. При этом добавляемый код должен не только безболезненно интегрироваться с уже существующим кодом. Необходимо иметь возможность запуска некоторых его фрагментов до начала выполнения основной программы, что позволяет осуществить предварительное связывание добавляемых объектов с объектами основной программы.

В целом же, на пути к эволюционной разработке программ можно отметить некоторые основные достижения:

– модули, содержащие код инициализации;
– пространства имен, размещаемые в нескольких единицах компиляции;
– классы, имеющие конструкторы и деструкторы;
– механизмы наследования и виртуализации;
– метапрограммирование;
– аспектно-ориентированное программирование;
– связывание через интерфейсы.

Существование различных подходов эволюционного формирования структуры программы неразрывно связано с моментами их использования в процессе подготовки

программы к решению требуемой задачи.

Этот процесс может автоматически поддерживаться различными инструментальными средствами. Помимо этого система может лишь предоставлять возможности написания кода, позволяющего программисту явным образом осуществлять его эволюционное расширение.

Окончательная модель предметной области, «зашитая» в разработанную программу, может сформироваться на одном из следующих этапов:

1) *Связывание добавляемых фрагментов до начала трансляции*. В этом случае используются макропроцессоры или препроцессоры, формирующие окончательный исходный текст программы. Последующая трансляция осуществляется по традиционной схеме.

2) *Связывание добавляемых понятий во время трансляции*. Подобная ситуация возникает в том случае, когда компилятор осуществляет формирование нужных программных объектов и производит дополнительные вычисления, обеспечивающие создание окончательной структуры модели предметной области. Этот вариант связывания обычно реализуется с использованием шаблонов.

3) *Связывание в процессе компоновки после трансляции*. Обычно сама трансляция осуществляется независимо для каждой единицы компиляции с формированием дополнительных данных для связывания. Последующая компоновка позволяет оформить окончательную структуру программы. В этом случае код, расширяющий существующую программу после трансляции должен предоставлять дополнительную информацию, обеспечивающую связывание. Ранее написанная программа должна поддерживать механизмы интеграции с дополнительным кодом.

4) *Связывание в процессе начальной загрузки*. Позволяет осуществить окончательную компоновку основной программы и расширяющих модулей во время загрузки и инициализации. В этом случае обеспечивается возможность выполнения дополнительного программного кода, отвечающего за связывание модулей.

5) *Явно задаваемое эволюционное связывание*. Обеспечивается программистом, который должен написать дополнительный код, обеспечивающий окончательное формирование структуры модели предметной области. Может осуществляться в любой из моментов выполнения программы после ее загрузки. Явно управляемое связывание позволяет гибко расширять программу за счет использования дополнительного кода, а не за счет инструментальной поддержки.

В настоящее время используются различные сочетания методов расширения и окончательного формирования структуры программы. Это позволяет решать многие задачи эволюционного проектирования.

Далее рассмотрим несколько используемых подходов и применяемых при этом программных объектов.

3.2. Тема 17: Базовые конструктивы процедурного и объектно-ориентированного подходов. Основные виды отношений между базовыми программными объектами. Конструирование агрегатов (8 часов).

3.2.1. Перечень и краткое содержание рассматриваемых вопросов:

1. Основные виды отношений между базовыми программными объектами

Несмотря на наличие расширяемых конструкций, обеспечивающих формирование общей структуры программы (модули, единицы компиляции, пространства имен), важную роль при разработке эволюционно расширяемых программ играют базовые понятия языков программирования, которые имеют разную реализацию в зависимости от применяемых парадигм.

В процедурном программировании в качестве таких базовых элементов используются данные и процедуры.

В объектно-ориентированном подходе основным элементов является класс, объединяющий данные и процедуры-методы воедино.

Отличие этих элементов ведет к различным принципам формирования кода и, следовательно, разным методам поддержки эволюционной разработки программного обеспечения в различных парадигмах программирования.

Особенности парадигм программирования проявляются в реализациях моделей состояния и поведения, а также отношений между этими понятиями, осуществляемых через такие элементарные программные объекты как данные и операции.

Абстрагирование от конкретных экземпляров достигается за счет введения понятий "*абстрактный тип данных*" и "*процедура*" (понятие "*функция*" используется как синоним процедуры).

Элементарные понятия используются для построения составных программных объектов путем объединения в агрегаты и разделения по категориям. Категорию называют иерархией типа "*is-a*". Она также трактуется как обобщение программных объектов.

Агрегаты и *обобщения* используются при конструировании композиций данных и процедур.

В каждой из существующих парадигм программирования вопросы такого конструирования композиций решаются по-своему, что и вносит определенные отличительные черты.

2. Конструирование агрегатов

Агрегация (агрегирование) – это абстрагирование, посредством которого один объект конструируется из других.

В связи с тем, что на самом нижнем уровне в программировании нами выделены две базовые абстракции (данные и процедуры), построение агрегатов может происходить следующими путями:

– агрегирование данных;

– агрегирование процедур;

– комбинированное агрегирование, обусловленное различными сочетаниями процедур и данных

В данном случае агрегирование рассматривается как конструирование новых программных объектов (данных, процедур и смешанных конструкций), из существующих. То есть, тело процедуры также можно рассматривать как агрегат, состоящий из команд и вызовов процедур.

Методы агрегирования

Агрегирование обеспечивает формирование программных объектов одним из способов:

– непосредственным включением;

– косвенным (ссылочным) связыванием, с применением наследования (расширения);

– образного восприятия.

Наиболее типичным является восприятие агрегата как единой абстракции, сформированной *непосредственным включением* используемых в нем программных объектов. В нашем сознании он видится как единый, монолитный ресурс, занимающий некоторое неразрывное.

Косвенное связывание позволяет формировать агрегаты из отдельных элементов, уже располагаемых в пространстве. Взаимосвязь осуществляется алгоритмическим заданием значений ссылок.

Спецификой является необходимость выполнения кода обеспечивающего конструирование объекта из разрозненных экземпляров абстракций. Однако этот код

выполняется только один раз, после чего работа с агрегатом осуществляется точно так же, как и в предыдущем случае.

В ряде случаев, когда элементы и агрегат создаются статически, инициализация связей может быть проведена во время трансляции программы.

Образный пример агрегата, построенного с применением косвенного связывания, представлен на рис. 1.

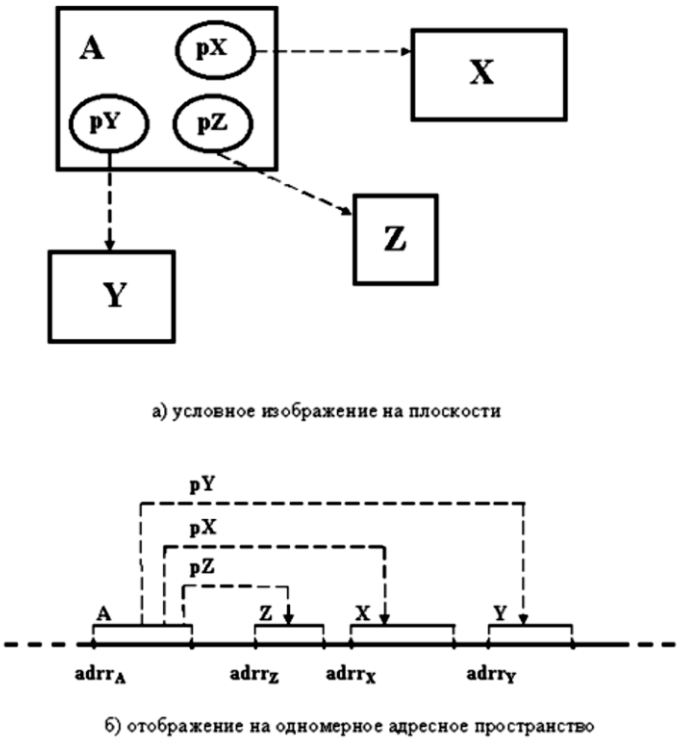

Рис. 1. Агрегат A из элементов X, Y, Z, построенный с использованием косвенного связывания

3.3. Тема 18: Различие парадигм при эволюционной разработке программ. Расширение обобщений. Расширение агрегатов (8 часов).

3.3.1. Перечень и краткое содержание рассматриваемых вопросов:

1. Расширение обобщений

Основным достоинством объектного обобщения является поддержка быстрого и безболезненного добавления новых альтернатив в уже написанный код при условии, что методы базового класса не изменяются. В той ситуации, где процедурный подход ведет к поиску и редактированию фрагментов программы, ООП довольствуется только созданием и легкой притиркой новых классов.

При разработке больших программных систем обработка расширяемых обобщений может осуществляться не одной сотней процедур, в каждую из которых потребуется внести изменения. А изменение написанного кода всегда связано с риском внести в программу дополнительные ошибки. Кроме этого, изменения, связанные с добавлением сведений о новой специализаций могут затронуть различные единицы компиляции (без изменения располагаемых в них программных объектов), что тоже ведет к дополнительным затратам.

ОО подход, в аналогичной ситуации, позволяет провести добавление круга практически без изменений уже написанного кода. При этом, все добавления, связанные с

новой фигурой могут осуществляться во вновь создаваемых единицах компиляции. Процесс добавления новой фигуры полностью аналогичен разработке уже созданных.

2. Расширение агрегатов

Кроме наращивания альтернатив, ОО обобщение прекрасно поддерживает эволюционное расширение агрегатов за счет построения производных классов на основе существующих. Применяя наследование, можно легко расширять внутреннюю структуру и функциональность объекта, предоставляемого клиентам, которым требуются дополнительные возможности. А сохранение возможностей старого интерфейса в новом классе обеспечивает прозрачную для старых клиентов подмену одного объекта другим.

Использование процедурного подхода не позволяет осуществлять такую подмену, так как создание нового агрегата сопровождается и созданием для его обработки новых процедур. В связи с тем, что обращение к агрегату осуществляется через формальный параметр определенного типа, приходится менять интерфейс клиента.

Независимо от техники и парадигм программирования создаются программные объекты и отношения между ними, с использованием таких понятий как агрегат и обобщение. При этом к одному и тому же конечному результату можно прийти различными путями. Необходимость использовать методы эволюционной разработки программного обеспечения ведет к пересмотру способов написания программ. Применение только одного подхода становится негибким и не позволяет удовлетворить требуемым критериям качества.

Процедурный подход обеспечивает гибкое добавление новых функций без изменения уже написанного кода. Однако изменение уже существующих данных ведет к значительным изменениям в обрабатывающих их процедурах. Объектно-ориентированная парадигма, напротив, позволяет легко добавлять новые объекты за счет распределения функциональности по разным классам. Применение смешанных мультипарадигменных подходов позволяет повысить гибкость и эффективность при разработке больших, эволюционно расширяемых программных систем.