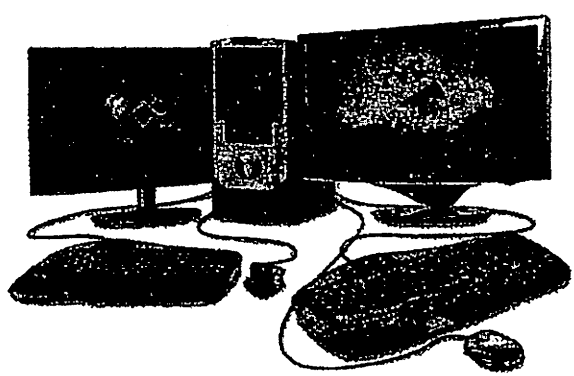
МИНИСТЕРСТВО ПО РАЗВИТИЮ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И КОММУНИКАЦИЙ РЕСПУБЛИКИ УЗБЕКИСТАН
ТАШКЕНТСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ ИМЕНИ МУХАММАДА АЛ-ХОРАЗМИЙ

Факультет «Компьютерный инжиниринг»
Кафедра «Компьютерные системы»

Н.А.Сайфуллаева, Р.Э. Яхшибоев, И.Х. Жабборов, С.Б. Довлетова

«АРХИТЕКТУРА КОМПЬЮТЕРА»
МЕТОДИЧЕСКИЕ УКАЗАНИЯ
ПО ВЫПОЛНЕНИЕ ПРАКТИЧЕСКИХ РАБОТ

для студентов по направлении
"5330500 – Компьютерный инжиниринг («Компьютерный инжиниринг»)"
5350100 – Телекоммуникационные технологии («Телекоммуникации, Телерадиовещание, Мобильные системы»)
ТУИТ имени Мухаммада аль-Хоразмий



Ташкент – 2022

УДК 004.7
Авторы: Сайфуллаева Н.А., Яхшибоев Р.Э., Жабборов Х.И., Довлетова С.Б.

«Компьютерные системы» / ТУИТ, 91с. Ташкент, 2022г.

Методические указания представляют собой указания для получения базовых знаний по изучению, архитектуру современных компьютеров, микросхем процессоров, принципов работы компьютерных шин, элементов памяти и организации памяти.

А также изучается работа процессоров и арифметико – логических устройств. Описываются основные логические элементы – вентили, булевых функции, микросхемы ввода – вывода и интерфейсы. Приводятся комбинаторные и арифметические схемы построение на основе логических элементов. Рассматриваются архитектура и выходы микросхем центральных процессоров Intel, AMD и VIA.

Даны описания архитектуры компьютера и организацию памяти.

Методическое указание предназначено студентам по "5330500 Компьютерный инжиниринг («Компьютерный инжиниринг»)" и 5350100 – Телекоммуникационные технологии («Телекоммуникации, Телерадиовещание, Мобильные системы») направлениям Ташкентский университет информационных технологии имени Мухаммада ал-Хоразмий.

Рецензенты:

Базарбаев М.И - К.ф-м.н, доцент, заведующий кафедры "Информатика и биофизика" Ташкентская медицинская академии

Ганиев А.А - К.т.н., доцент кафедры «Информационной безопасности» ТУИТ имени Мухаммада ал-Хоразмий

Практическое занятие № 1

Тема: Организация общей структуры компьютерной системы.

Цели работы: Изучения общей структуры компьютерной системы

Теоретическая часть

Структура компьютера — это совокупность его функциональных элементов и связей между ними. Элементами могут быть самые различные устройства — от основных логических узлов компьютера до простейших схем. Структура компьютера графически представляется в виде структурных схем, с помощью которых можно дать описание компьютера на любом уровне детализации.

Структурная организация компьютерной системы - совокупность операционных блоков (устройств) и их взаимосвязей, обеспечивающая реализацию спецификаций, заданных архитектурой компьютера

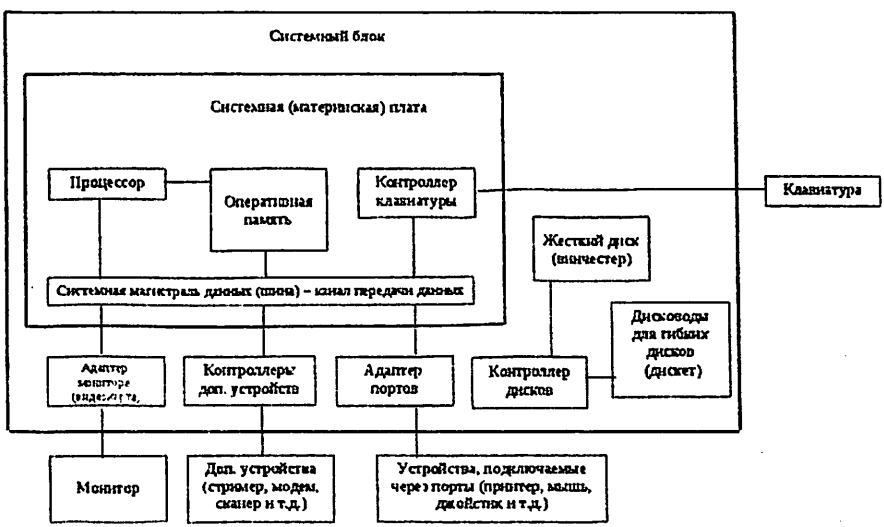


Рис.1 – Системный блок

Современный компьютер общего назначения состоит из одного или нескольких процессоров и множества контроллеров, связанных общей шиной, которая обеспечивает доступ к блоку памяти

Центральный процессор и контроллеры периферийного оборудования могут работать параллельно, взаимодействуя не непосредственно, а через

основную (оперативную) память (ОП). Для обеспечения корректного доступа к памяти используется соответствующий контроллер.

Функционирование компьютера после включения питания начинается с запуска программы первоначальной загрузки (bootstrap), которая инициализирует основные аппаратные блоки компьютера (регистры центрального процессора, контроллеры периферийного оборудования, накопители памяти), а затем загружает ядро ОС (operating system kernel) и передает ему управление. Обычно эта программа располагается в постоянном запоминающем устройстве (ПЗУ – Read Only Memory, ROM), возможно, электрически стираемом и перепрограммируемом (EEPROM).

Дальнейшее функционирование ОС осуществляется как реакция на события, происходящие в компьютере. Наступление того или иного события сигнализируется прерыванием (interrupt). Состояние устройства, работа которого прерывается, должно быть сохранено, только после этого можно производить обработку данного прерывания. Каждое прерывание обрабатывается соответствующим обработчиком прерывания (interrupt handler), входящим в состав ОС. После завершения обработки прерывания состояние прерванного устройства восстанавливается, чтобы можно было продолжить работу.

Источниками прерываний могут быть как аппаратура, так и программы. Аппаратура «сообщает» о прерывании асинхронно (в любой момент времени) путем пересылки в центральный процессор через общую шину сигнала прерывания. Программа «сообщает» о прерывании путем выполнения операции, называемой системным вызовом (system call).
Примеры событий, вызывающих прерывания:

- деление на ноль;
- переполнение;
- неправильное обращение к памяти;
- запрос на системное обслуживание;
- завершение операции ввода-вывода.

Механизмы индикации и обработки прерывания являются важной частью архитектуры компьютера. Хотя эти механизмы и отличаются в различных компьютерах, но главные их функции являются общими для всех типов компьютеров.

Главные функции механизма прерываний:

- распознавание/классификация прерывания;
- передача управления соответствующему обработчику прерывания;
- корректное возвращение к прерванной программе.

Переход от прерываемой программы к обработчику прерывания и обратно должен выполняться как можно быстрее, чтобы существенно не уменьшать производительность системы. Одним из быстрых методов распознавания прерывания (определения адреса требуемого обработчика) является использование вектора прерываний (interrupt vector).Вектор прерываний обычно хранится в начале адресного пространства основной (оперативной) памяти.

Обычно запрещается прерывание обработчика прерывания. Однако в некоторых ОС прерывания снабжаются приоритетами. Таким образом, работа обработчика прерывания с более низким приоритетом может быть прервана, если произошло прерывание с более высоким приоритетом.

Одним из способов, позволяющих некоторому устройству проверить состояние другого, независимо работающего устройства, является опрос; например, одно устройство может периодически проверять, находится ли другое устройство в определенном состоянии, и если нет, то продолжать свою работу.

Структура памяти

Программа может выполняться компьютером только в том случае, если она размещена в оперативной (основной) памяти. Оперативная память – единственная большая область памяти, к которой компьютер может обращаться непосредственно. Это массив слов или байтов, размером от сотен тысяч до сотен миллионов. Каждое слово имеет свой собственный адрес.

Обмен осуществляется с помощью команды загрузки содержимого памяти в регистр и выгрузки содержимого регистра в память. Кроме того, центральный процессор автоматически выбирает команды из ОП для выполнения.

Обычно, например, в компьютерах с фон Неймановской архитектурой, команда выбирается из ОП и размещается в специальном регистре команды. Затем команда дешифруется, а операнды, если они есть, выбираются из ОП и размещаются в нескольких внутренних регистрах. После того как действия над операндами выполнены, результат может быть записан обратно в основную память.

В идеале хотелось бы, чтобы программы и данные находились в основной памяти. Однако постоянно хранить их там невозможно по следующим двум причинам:

- основная память обычно слишком мала для размещения всех необходимых программ и данных;
- содержимое основной памяти теряется при отключении питания.

Поэтому в большинстве компьютеров имеется так называемая вторичная память как продолжение основной, то есть внешние запоминающие

устройства. Главное назначение вторичной памяти – длительное хранение больших массивов данных.

В качестве накопителя вторичной памяти обычно используется магнитный диск, на котором размещаются как программы, так и данные.

Все разнообразие запоминающих устройств компьютера может быть организовано в иерархию по убыванию времени доступа, возрастанию цены и увеличению емкости

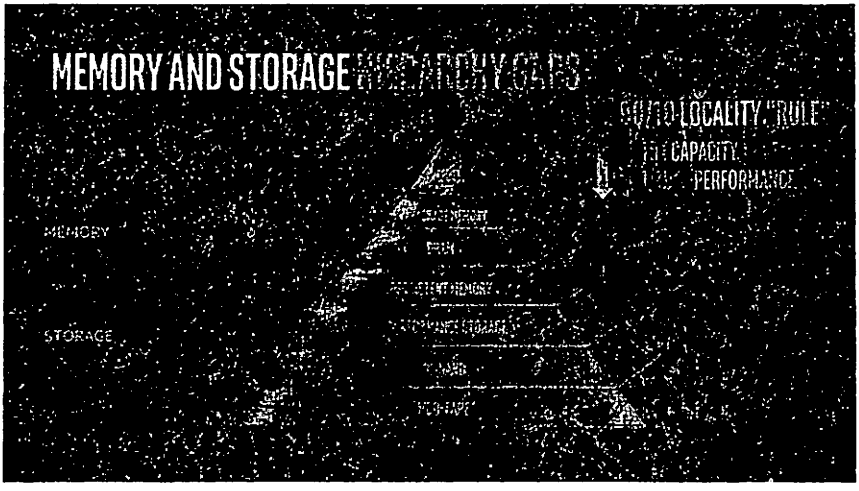


Рис.2 – Иерархия памяти

Многоуровневую схему используют следующим образом. Информация, которая находится в памяти верхнего уровня, обычно хранится также на более низких уровнях. Если процессор не обнаруживает нужную информацию на некотором уровне, он начинает искать ее на нижележащих уровнях. Когда нужная информация найдена, она переносится в более быстрые уровни.

Обычно в центральной части сосредоточены регистры процессора (CPU registers), основная память (main memory) и кэш-память (cashe memory).

Система команд центрального процессора содержит команды, позволяющие работать с регистрами и основной памятью. Работа с кэш-памятью осуществляется на аппаратном уровне контроллером кэш-памяти. Для работы с вторичной (внешней) памятью нужна программа. Таким образом, программно-доступными являются основная и вторичная память, а также регистры центрального процессора. Кэш-память программно недоступна.

Метод доступа к адресуемым элементам памяти центральной части называется произвольным (random access). Это означает, что все элементы доступа равнодоступны и для доступа к ним не требуется просмотра других элементов (direct access), и время доступа не зависит от адреса (arbitrary access).

На практике контроллер открывает доступ к содержимому регистра/ячейки, используя в качестве входной информации адрес регистра/ячейки. Причем последовательность адресов на вход контроллера может следовать в произвольном порядке.

Адресное пространство буферных запоминающих устройств ввода-вывода может быть частью адресного пространства основной памяти. Например, контроллер видеодисплея компьютеров IBM PC и Apple Macintosh.

Время доступа к ячейкам основной памяти в несколько раз больше, чем время доступа к регистрам. Поэтому часто, с целью уменьшения среднего времени доступа к памяти центральной части в архитектуру компьютера включают блок памяти, называемый кэш-память. Кэш-память обладает временем доступа равным или соизмеримым со временем доступа к регистрам.

В кэш-память облачно копируется информация из основной памяти. Эффективность кэш-памяти с точки зрения увеличения производительности во многом зависит от стратегии копирования.

Вначале центральный процессор проверяет наличие требуемых данных в кэш-памяти, а затем в основной. Хорошая стратегия копирования обеспечивает нахождение информации в кэш-памяти в 80% - 99% случаев обращения процессора к памяти.

Необходимость вторичной памяти обусловлена двумя причинами:

- емкость основной памяти недостаточна для хранения всех программ и данных;
- основная память энергозависима и ее содержимое теряется после отключения напряжения питания.

В качестве накопителя вторичной памяти обычно используется магнитный диск. Накопитель на магнитном диске обычно состоит из нескольких дисков, покрытых слоем магнитного материала и расположенных на общей оси вращения.

В процессе работы диски вращаются с постоянной скоростью. Информация наносится на поверхность магнитного материала побитно в виде концентрических окружностей, называемых дорожками (tracks). Любая

дорожка содержит одинаковое количество информации. Дорожки разделены на секторы (sectors). Сектор является наименьшей адресуемой порцией информации для магнитного диска.

Запись/считывание данных осуществляется без остановки диска с помощью универсальных головок чтения/записи. Головки устанавливаются на соответствующую дорожку с помощью механизма доступа.

Накопители на магнитном диске отличаются количеством поверхностей, диаметром дисков, возможностью легкого извлечения накопителя из компьютера.

Структура ввода-вывода

В состав компьютера общего назначения входят центральный процессор и множество контроллеров, которые соединены общей шиной. Контроллеры периферийного оборудования специализированы по типу оборудования и обычно один контроллер управляет устройством одного типа. Однако иногда конструируются универсальные контроллеры, предназначенные для работы с несколькими разнотипными устройствами.

Например, контроллер SCSI (Small Computer System Interface) позволяет подключать до семи различных устройств. Каждый контроллер снабжен памятью, включающей буферное запоминающее устройство, регистр команды, регистр состояния. Контроллер должен обладать способностью перемещать данные между периферийным устройством, которым он управляет, и локальной буферной памятью. Обычно в операционной системе имеется соответствующий драйвер (driver) для каждого контроллера. Драйвер умеет напрямую работать с устройством и предоставляет унифицированный интерфейс для остальной части операционной системы.

Данные, выводимые на носитель информации или вводимые с носителя информации, предварительно накапливаются в буфере. Размер буфера различен для разных устройств и определяется спецификой устройства. Контроллер выполняет приказы (команды) центрального процессора, поступающие в регистр команд.

Например, получив команду прочитать данные с носителя, контроллер вырабатывает последовательность управляющих сигналов, в результате чего поверхность носителя перемещается, информация считывается, преобразуется и записывается в буфер.

Существует два метода взаимодействия центрального процессора и контроллера: синхронный и асинхронный. При синхронном методе процесс, запрашивающий операцию ввода-вывода, после прерывания ожидает

завершения операции контроллером. При асинхронном методе – после передачи в контроллер команды ввода-вывода, центральный процессор и контроллер работают раздельно.

При синхронном методе ожидание процессором завершения операции ввода-вывода означает, что он работает «вхолостую». Наиболее простой способ заставить процессор работать «вхолостую» заключается в организации «вечного» цикла. Этот цикл продолжается до тех пор, пока не получено прерывание. Некоторые компьютеры включают в свою систему команд специальную команду wait, переводящую процессор в «холостой» режим работы. Главное преимущество синхронного метода состоит в легкости определения контроллера, требующего прерывания.

Асинхронный метод предполагает одновременную работу процессора и одного или нескольких контроллеров, что повышает эффективность использования оборудования, однако усложняет механизм прерываний. При асинхронном методе, например, возможна ситуация, когда несколько различных процессов требуют операции ввода-вывода с одним и тем же устройством.

Контрольные вопросы

1. Что такое структура компьютера?
2. Приведите примеры событий, вызывающих прерывания. Поясните главные функции механизма прерываний.
3. Опишите иерархию запоминающих устройств компьютера. Что означает произвольный метод доступа к адресуемым элементам памяти?
4. Зачем нужна кэш-память?
5. Опишите различия между синхронным и асинхронным методами взаимодействия центрального процессора и контроллера.

Задание. Каждый студент должен описать структуру своего персонального компьютера

Содержание отчёта

1. Использовать листы формата А4
2. Использовать шрифта Times New Roman, размер кегля 14
3. Интервал 1,5
4. Написать тему, цель и задачи практического работы
5. Выполнить задание
6. Написать выводы и ответить письменно на вопросы.

Практическое занятие №2

Тема: Архитектурные типы многоядерных процессоров
Цель работы: Изучение архитектурных типов многоядерных процессоров.

Теоретическая часть

Компьютерная техника развивается быстрыми темпами. Вычислительные устройства становятся мощнее, компактнее, удобнее, однако в последнее время повышение производительности устройств стало большой проблемой. В 1965 году Гордон Мур (один из основателей Intel) пришёл к выводу, что «количество транзисторов, размещаемых на кристалле интегральной схемы, удваивается каждые 24 месяца».

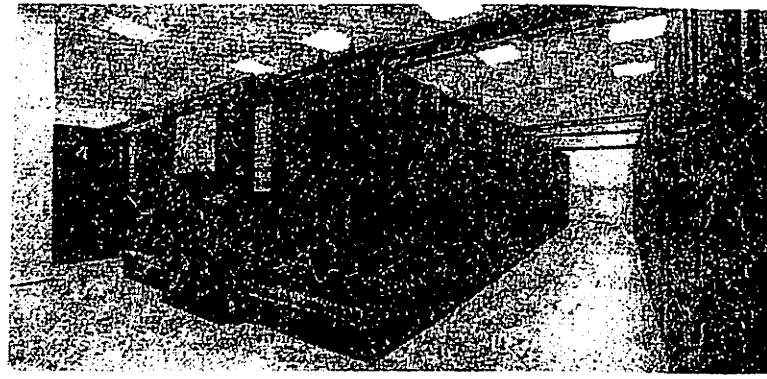


Рис.3 – Многоядерные процессоры

Первые разработки в области создания многопроцессорных систем начались в 70-х годах. Длительное время производительность привычных одноядерных процессоров повышалась за счёт увеличения тактовой частоты (до 80% производительности определяла только тактовая частота) с одновременным увеличением числа транзисторов на кристалле. Фундаментальные законы физики остановили этот процесс: чипы стали перегреваться, технологический стал приближаться к размерам атомов кремния. Все эти факторы привели к тому, что:

- увеличились токи утечки, вследствие чего повысилось тепловыделение и потребляемая мощность.

• процессор стал намного «быстрее» памяти. Производительность снижалась из-за задержки обращения к оперативной памяти и загрузке данных в кэш.

• возникает такое понятие как «фон-неймановское узкое место». Оно означает неэффективность архитектуры процессора при выполнении какой-либо программы.

Многопроцессорные системы (как один из способов решения проблемы) не получили широко применения, так как требовали дорогостоящих и сложных в производстве многопроцессорных материнских плат. Исходя из этого, производительность повышалась иными путями. Эффективной оказалась концепция многопоточности – одновременная обработка нескольких потоков команд.

Hyper-Threading Technology (HTT) или технология сверхпоточной обработки данных, позволяющая процессору на одном ядре выполнять несколько программных потоков. Именно HTT по мнению многих специалистов стала предпосылкой для создания многоядерных процессоров. Выполнение процессором одновременно несколько программных потоков называется параллелизмом на уровне потоков (TLP –thread-level parallelism).

Для раскрытия потенциала многоядерного процессора исполняемая программа должна задействовать все вычислительные ядра, что не всегда достижимо. Старые последовательные программы, способные использовать лишь одно ядро, теперь уже не будут работать быстрее на новом поколении процессоров, поэтому в разработке новых микропроцессоров всё большее участие принимают программисты.

Архитектура в широком смысле – это описание сложной системы, состоящей из множества элементов.

В процессе развития полупроводниковые структуры (микросхемы) эволюционируют, поэтому принципы построения процессоров, количество входящих в их состав элементов, то, как организовано их взаимодействие, постоянно изменяются. Таким образом, CPU с одинаковыми основными принципами строения, принято называть процессорами одной архитектуры. А сами такие принципы называют архитектурой процессора (или микроархитектурой).

Микропроцессор (или процессор) – это главный компонент компьютера. Он обрабатывает информацию, выполняет программы и управляет другими устройствами системы. От мощности процессора зависит, насколько быстро будут выполняться программы.

Ядро - основа любого микропроцессора. Оно состоит из миллионов транзисторов, расположенных на кристалле кремния. Микропроцессор

разбит на специальные ячейки, которые называются регистрами общего назначения (РОН). Работа процессора в общей сложности состоит в извлечении из памяти в определённой последовательности команд и данных и их выполнении. Кроме того, ради повышения быстродействия ПК, микропроцессор снабжён внутренней кэш-памятью. Кэш-память - это внутренняя память процессора, используемая в качестве буфера (для защиты от перебоев со связью с оперативной памятью).

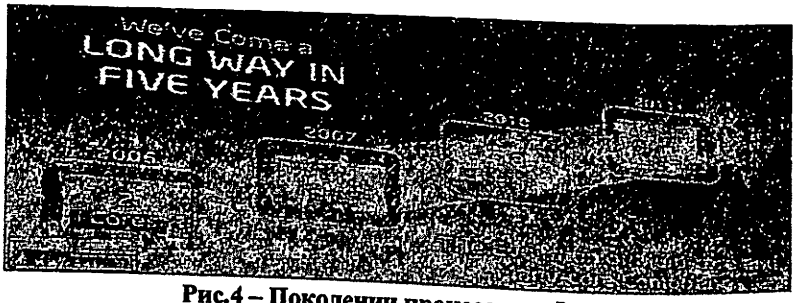


Рис.4 – Поколении процессоров Intel

Процессоры Intel, используемые в IBM – совместимых ПК, насчитывают более тысячи команд и относятся к процессорам с расширенной системой команд – CISC-процессорам (CISC –Complex Instruction Set Computing).

Высокопроизводительные вычисления. Параллелизм. Темпы развития вычислительной техники легко проследить: от ENIAC (первый электронный цифровой компьютер общего назначения) с производительностью в несколько тысяч операций в секунду до суперкомпьютера Tianhe-2 (1000 триллионов операций с плавающей запятой в секунду). Это означает, что скорость вычислений увеличилась в триллион раз за 60 лет. Создание высокопроизводительных вычислительных систем – одна из самых сложных научно-технических задач. При том, что скорость вычислений технических средств выросла всего лишь в несколько миллионов раз, общая скорость вычислений выросла в триллионы раз. Этот эффект достигнут за счёт применения параллелизма на всех стадиях вычислений. Параллельные вычисления требуют поиска рационального распределения памяти, надёжных способов передачи информации и координации вычислительных процессов.

Симметрическая мультипроцессорность. Symmetric Multiprocessing (сокращённо SMP) или симметрическое мультипроцессирование – это особая архитектура мультипроцессорных систем, в которой несколько процессоров

имеют доступ к общей памяти. Это очень распространённая архитектура, достаточно широко используемая в последнее время.

При применении SMP в компьютере работает сразу несколько процессоров, каждый над своей задачей. SMP система при качественной операционной системе рационально распределяет задачи между процессорами, обеспечивая равномерную нагрузку на каждый из них. Однако возникает проблема к обращению памяти, ведь даже однопроцессорным системам требуется на это относительно большое время. Таким образом, обращение к оперативной памяти в SMP происходит последовательно: сначала один процессор, затем второй.

В силу перечисленных выше особенностей, SMP-системы применяется исключительно в научной сфере, промышленности, бизнесе, крайне редко в рабочих офисах. Кроме высокой стоимости аппаратной реализации, такие системы нуждаются в очень дорогом и качественном программном обеспечении, обеспечивающем многопоточное выполнение задач. Обычные программы (игры, текстовые редакторы) не будут эффективно работать в SMP-системах, так как в них не предусмотрена такая степень распараллеливания. Если адаптировать какую-либо программу для SMP-системы, то она станет крайне неэффективно работать на однопроцессорных системах, что приводит к необходимости создание нескольких версий одной и той же программы для разных систем. Исключение составляет, например, программа ABLETON LIVE (предназначена для создания музыки и подготовка Dj-сетов), имеющая поддержку мультипроцессорных систем. Если запустить обычную программу на мультипроцессорной системе, она всё же станет работать немного быстрее, чем в однопроцессорной. Это связано с так называемым аппаратным прерыванием (остановка программы для обработки ядром), которое выполняется на другом свободном процессоре.

SMP-система (как и любая другая, основанная на параллельных вычислениях) предъявляет повышенные требования к такому параметру памяти, как полоса пропускания шины памяти. Это зачастую ограничивает количество процессоров в системе (современные SMP- системы эффективно работают вплоть до 16 процессоров).

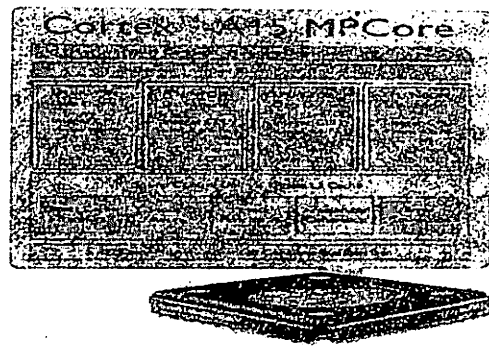


Рис.5 – Процессор Cortex A15 MPCore

Так как у процессоров общая память, то возникает необходимость рационального её использования и согласования данных. В мультипроцессорной системе получается так, что несколько кэшей работают для разделяемого ресурса памяти. Cache coherence (когерентность кэша) – свойство кэша, обеспечивающее целостность данных, хранящихся в индивидуальных кэшах для разделяемого ресурса. Данное понятие – частный случай понятия когерентности памяти, где несколько ядер имеют доступ к общей памяти (повсеместно встречается в современных многоядерных системах)[4]. Если описать данные понятия в общих чертах, то картина будет следующей: один и тот же блок данных может быть загружен в разные кэши, где данные обрабатываются по-разному.

Если не будут использованы какие-либо уведомления об изменении данных, то возникнет ошибка. Когерентность кэша призвана для разрешения таких конфликтов и поддержки соответствия данных в кэшах.

SMP-системы являются подгруппой MIMD (multi in-struction multi data - вычислительная система со множественным потоком команд и множественным потоком данных) классификации вычислительных систем по Флинну (профессор Стэнфордского университета, сооснователь Palyn Associates). Согласно данной классификации, практически все разновидности параллельных систем можно отнести к MIMD.

Разделение многопроцессорных систем на типы происходит на основе разделения по принципу использования памяти. Этот подход позволил различить следующие важные типы

многопроцессорных систем – multiprocessors (мультипроцессорные системы с общей разделяемой памятью) и multicomputers (системы с раздельной памятью). Общие данные, используемы при параллельных вычислениях требуют синхронизации. Задача синхронизация данных – одна

из самых важных проблем, и её решение при разработке многопроцессорных и многоядерных и, соответственно, необходимого программного обеспечения является приоритетной задачей инженеров и программистов. Общий доступ к данным может быть произведён при физическом распределении памяти. Этот подход называется неоднородным доступом к памяти (non-uniform memory access или NUMA).

Среди данных систем можно выделить:

- системы, где только индивидуальная кэш-память процессоров используется для представления данных (cache-only memory architecture).
- системы с обеспечением когерентности локальных кэшей для различных процессоров (cache-coherent NUMA).
- системы с обеспечением общего доступа к индивидуальной памяти процессоров без реализации на аппаратном уровне когерентности кэша (non-cache coherent NUMA).

Упрощение проблемы создания мультипроцессорных систем достигается использованием распределённой общей памяти (distributed shared memory), однако этот способ приводит к ощутимому повышению сложности параллельного программирования.

Одновременная многопоточность. Исходя из всех вышеперечисленных недостатков симметрической мультипроцессорности, имеет смысл разработка и развитие других способов повышения производительности. Если проанализировать работу каждого отдельного транзистора в процессоре, можно обратить внимание на очень интересный факт – при выполнении большинства вычислительных операций задействуются далеко не все компоненты процессора (согласно последним исследованиям – около 30% всех транзисторов). Таким образом, если процессор выполняет, скажем, несложную арифметическую операцию, то большая часть процессора простаивает, следовательно, её можно использовать для других вычислений. Так, если в данный момент процессор выполняет вещественные операции, то в свободную часть можно загрузить целочисленную арифметическую операцию. Чтобы увеличить нагрузку на процессор, можно создать спекулятивное (или опережающее) выполнение операций, что требует большого усложнения аппаратной логики процессора. Если в программе заранее определить потоки (последовательности команд), которые могут выполняться независимо друг от друга, то это заметно упростит задачу (данный способ легко реализуется на аппаратном уровне). Эта идея, принадлежащая Дину Тулсену (разработана им в 1955 г в университете Вашингтона), получила название одновременной многопоточности (simul-taneous multithreading). Позднее она была развита компанией Intel под названием гиперпоточности (hyper threading). Так, один

процессор, выполняющий множество потоков, воспринимается операционной системой Windows как несколько процессоров. Использование данной технологии опять-таки требует соответствующего уровня программного обеспечения. Максимальный эффект от применения технологии многопоточности составляет около 30%.

Многоядерность. Технология многопоточности – реализация многоядерности на программном уровне. Дальнейшее увеличение производительности, как всегда, требует изменений в аппаратной части процессора. Усложнение систем и архитектур не всегда оказывается действенным. Существует обратное мнение: «всё гениальное – просто!». Действительно, чтобы повысить производительность процессора вовсе необязательно повышать его тактовую частоту, усложнять логическую и аппаратную составляющие, так как достаточно лишь провести рационализацию и доработку существующей технологии. Такой способ весьма выгоден – не нужно решать проблему повышения тепловыделения процессора, разработку нового дорогостоящего оборудования для производства микросхем. Данный подход и был реализован в рамках технологии многоядерности – реализация на одном кристалле нескольких вычислительных ядер. Если взять исходный процессор и сравнить прирост производительности при реализации нескольких способов повышения производительности, то очевидно, что применение технологии многоядерности является оптимальным вариантом.

Если сравнивать архитектуры симметричного мультипроцессора и многоядерного, то они окажутся практически идентичными. Кэш-память ядер может быть многоуровневой (локальной и общей, причём данные из оперативной памяти могут загружаться в кэш-память второго уровня напрямую). Исходя из рассмотренных достоинств многоядерной архитектуры процессоров, производители делают акцент именно на ней. Данная технология оказалась достаточно дешёвой в реализации и универсальной, что позволило вывести её на широкий рынок. Кроме того, данная архитектура внесла свои коррективы в закон Мура: «количество вычислительных ядер в процессоре будет удваиваться каждые 18 месяцев».

Если посмотреть на современный рынок компьютерной техники, то можно увидеть, что доминируют устройства с четырёх- и восьми- ядерными процессорами. Кроме того, производители процессоров заявляют, что в скором времени на рынке можно будет увидеть процессоры с сотнями вычислительных ядер. Как уже неоднократно говорилось ранее, весь потенциал многоядерной архитектуры раскрывается только при наличии качественного программного обеспечения. Таким образом, сфера

производства компьютерного «ожелеза» и программного обеспечения очень тесно связаны между собой.

Контрольные вопросы:

1. Что вы понимаете под термином многоядерность?
2. Что вы знаете о многопроцессорных системах 70-х годов?
3. Какую роль выполняют применение технологии SMP в компьютерах?
4. Расскажите о технологии многопоточности и о его развитии.
5. Перечислите архитектуры многоядерных процессоров и дайте определение каждому из них?

Задание. Сравнивать поколении процессоров Intel и AMD.

Содержание отчёта

1. Использовать листы формата А4
2. Использовать шрифта Times New Roman, размер кегля 14
3. Интервал 1,5
4. Написать тему, цель и задачи практического работы
5. Выполнить задание
6. Написать выводы и ответить письменно на вопросы.

Практическое занятие № 3

Тема: Изучение кластерных систем

Цели работы: Изучение кластерных систем и их построение.

Теоретическая часть

Кластерный подход к созданию суперкомпьютеров

Кластеры стали применяться в сфере высокопроизводительных вычислений сравнительно недавно. До конца 80-х практически все суперкомпьютеры представляли собой большой массив соединенных между собой процессоров. Подобные разработки чаще всего были уникальными и имели огромную стоимость не только приобретения, но и поддержки. Поэтому в 90-х годах все более широкое распространение стали получать кластерные системы, которые в качестве основы используют недорогие однотипные вычислительные узлы.

Основными достоинствами кластерного подхода являются именно дешевизна и легкая расширяемость. Цены на системы кластерного типа стремительно падают, а некоторые модели уже сейчас доступны для одиночных исследователей. К недостаткам же можно отнести сложность создания и отладки эффективных параллельных программ для систем с разделенной памятью.

Однако в последнее время развиваются инструменты, облегчающие написание параллельных программ, что способствует все большему распространению кластеров. Как уже было сказано выше, кластеры представляют собой системы с разделенной памятью, поэтому типичное параллельное приложение представляет собой совокупность нескольких процессов, исполняемых на разных вычислительных узлах и взаимодействующих по сети. В принципе, разработчик может полностью взять на себя программирование распределенного приложения и самостоятельно реализовать общение по сети на основе сокетов, например.

Однако в настоящее время существует довольно большое число технологий, упрощающих создание параллельных приложений для кластеров: MPI , PVM, HPF и другие.

Эти технологии существуют уже достаточно продолжительное время, за которое они доказали свою состоятельность и легли в основу огромного числа параллельных приложений. Кластеры стали фактическим стандартом в области высокопроизводительных вычислений, и можно с большой долей

уверенности сказать, что этот подход будет актуален всегда: сколь бы совершенен не был один компьютер, кластер из узлов такого типа справится с любой задачей гораздо быстрее.

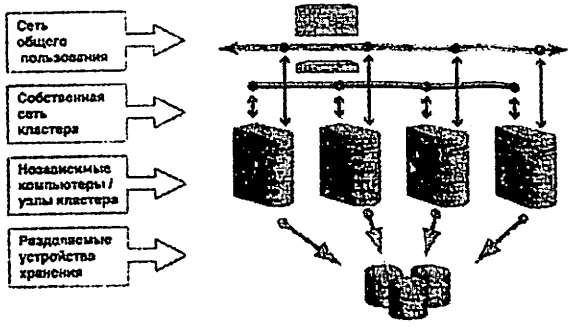


Рис.6 – Кластерный подход

Назначение систем управления кластерами

Суперкомпьютеры (и кластеры как подкласс) приобретаются для решения вычислительно трудоемких задач: моделирование климата, финансовые расчеты, генная инженерия, моделирование физических процессов, химических реакций и так далее. Сложность экспериментов такова, что могут потребоваться годы расчетов на одном процессоре, поэтому исследователи активно применяют распараллеливание. Как результат, колоссальные по своей сложности задачи могут быть решены «всего» за недели.

Однако при массовом запуске параллельных приложений на кластерах возникают новые сложности, которые связаны с тем, что чаще всего приходится иметь дело с большой серией экспериментов, каждый из которых имеет уникальные потребности относительно аппаратных ресурсов. В такой ситуации нужно очень умело распоряжаться узлами кластера, стараясь распределить нагрузку максимально равномерно.

Необходимо находить «зазоры» в расписании между тяжелыми задачами и стараться запускать между ними более легкие, которые должны решаться на оставшихся свободных (или просто наименее загруженных) узлах. Нельзя забывать также, что задачи могут иметь различный приоритет и между запусками приложений могут существовать зависимости, что накладывает определенные ограничения на расписание.

Все вопросы, связанные с составлением расписания запусков пользовательских приложений, ложатся на систему управления кластером.

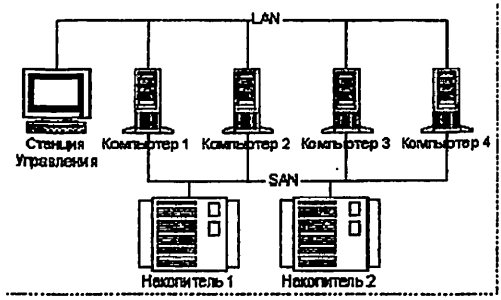
Помимо составления расписания система должна обеспечивать надежность функционирования кластера. Выход одного узла из строя не должен приводить к остановке работы всего кластера. В идеале система должна «на лету» обнаруживать и обрабатывать изменение состава вычислительных узлов. Кроме составления расписания есть еще ряд задач, которые ложатся на систему управления: это предоставление удаленного доступа к вычислительным ресурсам, предоставление средств администрирования и отслеживания состояния кластера и ряд других вопросов.

Основные требования к системам управления

Главной задачей системы управления кластерами является обеспечение максимально быстрого возврата результатов запуска при существующем аппаратном обеспечении и согласно установленным правилам использования вычислительных ресурсов.

Можно даже выдвинуть следующий критерий: хорошая система управления – это такая система, которую пользователь не замечает, имея возможность полностью сконцентрироваться на решаемой прикладной проблеме. Система должна максимально упрощать работу с вычислительным кластером и ускорять получение результатов экспериментов. Таким образом, с нашей точки зрения, можно выделить следующие три основополагающие требования к системам управления кластерами.

Производительность (высокая пропускная способность). Системы управления кластерами фактически представляют собой системы пакетной обработки, и поэтому их основная задача – обеспечить максимальное количество заданий, выполняемых кластером за единицу времени. Чем быстрее получается результат, чем быстрее эта обратная связь для исследователя, тем большее число экспериментов может быть проведено, что позволит глубже проникнуть в исследуемое явление или, например, разработать для коммерческой организации более выигрышную стратегию поведения на рынке. Для обеспечения высокой производительности в системах управления используется эффективное



априорное планирование и динамическое распределение нагрузки, основанное на миграции процессов.

Надежность (отказоустойчивость). Кластер должен функционировать непрерывно. Все аппаратные и программные сбои должны оперативно обрабатываться, после чего кластер должен продолжить выполнение заданий. Если сбой привел к остановке вычислений, то система должна уметь восстановить их некоторое промежуточное состояние и начать с него, а не с самого начала. Так, если эксперимент продолжался несколько недель, то остановка его при 90%-й готовности может оказаться катастрофой. Поэтому в развитых системах управления периодически создаются контрольные точки исполняющихся процессов, чтобы их при необходимости можно было перезапустить.

Удобство использования. Это достаточно размытое по смыслу требование является, однако, одним из наиболее важных. Система управления кластером должна быть гибкой и хорошо настраиваемой, а также простой в использовании.

В частности, система должна иметь несколько интерфейсов доступа, для того чтобы пользователь мог выбрать наиболее подходящий для его потребностей. Так, многие системы имеют одновременно GUI-интерфейс, интерфейс командной строки и интерфейс прикладного программирования (API). Администратор же должен иметь широкие возможности выбора алгоритмов и политик планирования, чтобы адаптировать систему к конкретным характеристикам кластера и классам решаемых задач.

Альтернативный подход к управлению кластерами

Системы управления кластерами решают задачу эффективного управления ресурсами кластера, но это можно делать различными способами, и в настоящее время используются два основных подхода:

· Создание распределенных окружений для кластерных вычислений (Distributed Cluster Computing Environments, DCCE); · Создание систем управления кластерами (Cluster Management Systems, CMS), или систем управления ресурсами и планирования (Resource Management and Scheduling, RMS). Идея первого подхода состоит в создании «кластерной» операционной системы, которая бы взяла на себя все вопросы, связанные с эффективным управлением кластером. Так, приложение не обязательно исполняется на том

узле, откуда был произведен запуск, а может мигрировать на менее загруженный узел.

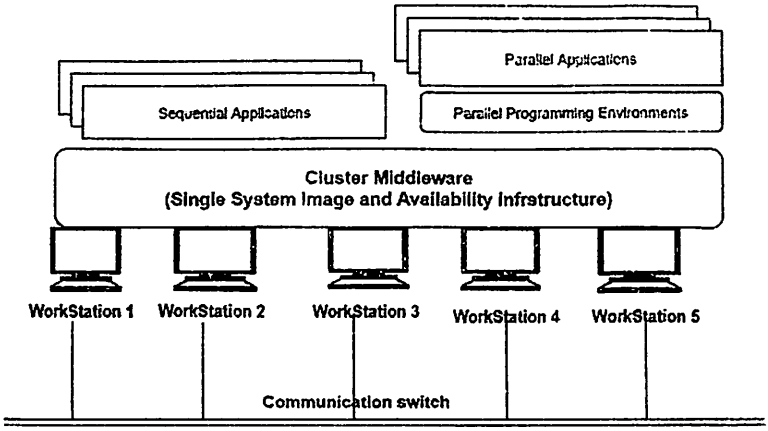


Рис.7 – Коммуникационный узел

Также на всем кластере используется общая файловая система, и пользователь может даже не задумываться над тем, на каком именно узле кластера находятся данные. Существует несколько реализаций подобных окружений: OpenMosix, Kerrighed , OpenSSI, Gluster и другие. Эти системы весьма удобны для пользователей, потому как работа в них фактически не отличается от работы в обычной операционной системе.

Недостаток подхода состоит в том, что он практически нереализуем без коррекции операционной системы. Именно поэтому все подобные окружения существуют только для UNIX-подобных систем с открытым кодом (Linux, FreeBSD).

Существует альтернативный способ, не связанный с модификацией операционной системы, – создание специализированных систем управления кластером, которые мы рассматриваем в настоящей работе. Как правило, используется следующий порядок работы: пользователь передает системе свое приложение, указывая параметры запуска, а система уже сама определяет в какой момент и на каких узлах будет запущено приложение. То есть системы управления кластерами устанавливаются поверх операционной

системы и являются приложениями пользовательского уровня. Оба изложенных подхода имеют свои достоинства и недостатки.

Заметим, что первый из них делает ставку на динамическое планирование и перепланирование (миграция процессов), в то время как в рамках второго подхода используются сложные алгоритмы планирования заданий. Стоит заметить, что в настоящее время более широко используется второй подход. Причина, по всей видимости, кроется в том, что он не требует изменения операционной системы.

Распределенные окружения в массе своей представляют экспериментальные образцы, на которых возможны проблемы, связанные с несовместимостью программного обеспечения. В такой ситуации понятен выбор пользователей, отдающих предпочтение системам управления кластерами, работающим на базе обычных операционных систем.

Обзор существующих систем управления

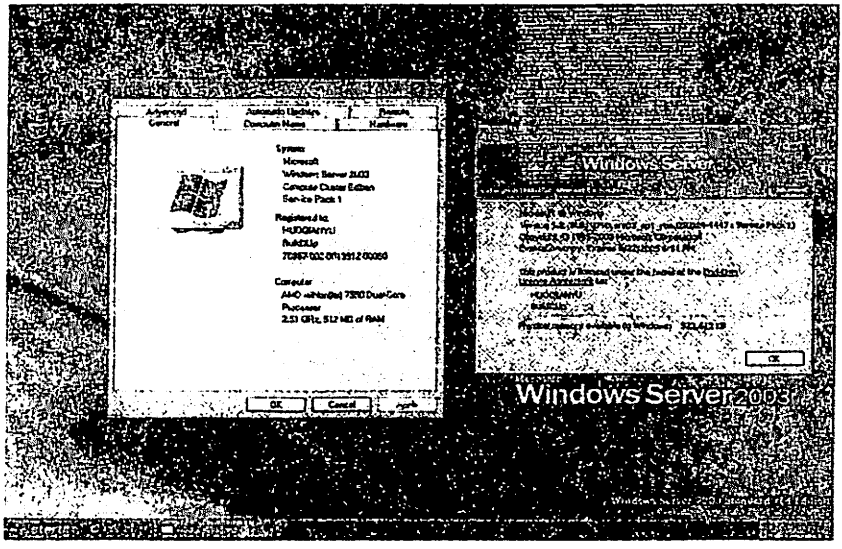


Рис.8 – Система управление ОС Windows

В настоящее время все сложнее найти систему управления, которая предлагается как самостоятельный продукт. И это вполне закономерно, поскольку сами по себе эти системы не представляют ценности, так как для использования кластера требуется большой пакет программного обеспечения. Именно поэтому большинство поставщиков кластерных решений включают системы управления в целые комплекты программноаппаратных средств. Так, Microsoft распространяет свою систему

управления вместе с 12 специальной версией операционной системы Windows Server 2003 и реализацией стандарта MPI2.

В настоящее время в мире существует достаточно большое число систем управления кластерами. Вот далеко не полный список наиболее популярных систем управления: Microsoft Compute Cluster Server 2003 , Condor , PBSPro, LSF, Sun Grid Engine. С появлением технологии GRID многие системы пошли в этом направлении и приобрели ряд новых полезных возможностей, как то работа в существенно гетерогенных и распределенных средах. Мы, однако, сконцентрируемся на тех возможностях, которые имеют отношение к исходному назначению систем управления – массовому запуску вычислительно трудоемких заданий.

Мы рассмотрим подробно две популярные системы управления кластерами: Microsoft Compute Cluster Server 2003 и Condor. Это рассмотрение интересно прежде всего тем, что две упомянутые системы имеют совершенно различные истории и внутреннее устройство, поэтому анализируя их можно сделать выводы об общих тенденциях развития систем управления кластерами.

Контрольные вопросы:

1.Для чего нам нужны кластерные системы?
2.Назовите системы управления кластерными системами.
3.Основные характеристики кластерной системы. Их преимущества и недостатки.
4.Приведите альтернативные способы управления кластерами
5.Нужно ли просматривать приоритеты задач в кластерных системах? Объясните свой ответ.

Содержание отчёта

1. Использовать листы формата А4
2. Использовать шрифта Times New Roman, размер кегля 14
3. Интервал 1,5
4. Написать тему, цель и задачи практического работы
5. Выполнить задание
6. Написать выводы и ответить письменно на вопросы.

Практическое занятие № 4

Изучение технологического процесса конвейерной обработки

Цели работы: Изучение понятия конвейерной обработки и оценка его производительности.

Теоретическая часть

Что такое конвейерная обработка

Разработчики архитектуры компьютеров издавна прибегали к методам проектирования, известным под общим названием "совмещение операций", при котором аппаратура компьютера в любой момент времени выполняет одновременно более одной базовой операции. Этот общий метод включает два понятия: параллелизм и конвейеризацию.

Хотя у них много общего и их зачастую трудно различать на практике, эти термины отражают два совершенно различных подхода. При параллелизме совмещение операций достигается путем воспроизведения в нескольких копиях аппаратной структуры. Высокая производительность достигается за счет одновременной работы всех элементов структур, осуществляющих решение различных частей задачи.

Конвейеризация (или конвейерная обработка) в общем случае основана на разделении подлежащей исполнению функции на более мелкие части, называемые ступенями, и выделении для каждой из них отдельного блока аппаратуры.

Так обработку любой машинной команды можно разделить на несколько этапов (несколько ступеней), организовав передачу данных от одного этапа к следующему.

При этом конвейерную обработку можно использовать для совмещения этапов выполнения разных команд. Производительность при этом возрастает благодаря тому, что одновременно на различных ступенях конвейера выполняются несколько команд. Конвейерная обработка такого рода широко применяется во всех современных быстродействующих процессорах.

Простейшая организация конвейера и оценка его производительности

Для иллюстрации основных принципов построения процессоров мы будем использовать простейшую архитектуру, содержащую 32 целочисленных регистра общего назначения (R0,...,R31), 32 регистра плавающей точки (F0,...,F31) и счетчик команд PC. Будем считать, что набор команд нашего процессора включает типичные арифметические и логические

операции, операции с плавающей точкой, операции пересылки данных, операции управления потоком команд и системные операции. В арифметических командах используется трехадресный формат, типичный для RISC-процессоров, а для обращения к памяти используются операции загрузки и записи содержимого регистров в память.

Выполнение типичной команды можно разделить на следующие этапы:

- выборка команды - IF (по адресу, заданному счетчиком команд, из памяти извлекается команда);
- декодирование команды / выборка операндов из регистров - ID;
- выполнение операции / вычисление эффективного адреса памяти - EX;
- обращение к памяти - MEM;
- запоминание результата - WB.

На схеме простейшего процессора, выполняющего указанные выше этапы выполнения команд без совмещения. Чтобы конвейеризовать эту схему, мы можем просто разбить выполнение команд на указанные выше этапы, отведя для выполнения каждого этапа один такт синхронизации, и начинать в каждом такте выполнение новой команды. Естественно, для хранения промежуточных результатов каждого этапа необходимо использовать регистровые станции.

На схеме процессора с промежуточными регистровыми станциями, которые обеспечивают передачу данных и управляющих сигналов с одной ступени конвейера на следующую. Хотя общее время выполнения одной команды в таком конвейере будет составлять пять тактов, в каждом такте аппаратура будет выполнять в совмещенном режиме пять различных команд.

Работу конвейера можно условно представить в виде сдвинутых во времени схем процессора. Этот рисунок хорошо отражает совмещение во времени выполнения различных этапов команд. Однако чаще для представления работы конвейера используются временные диаграммы, на которых обычно изображаются выполняемые команды, номера тактов и этапы выполнения команд.

Конвейеризация увеличивает пропускную способность процессора (количество команд, завершающихся в единицу времени), но она не сокращает время выполнения отдельной команды.

В действительности, она даже несколько увеличивает время выполнения каждой команды из-за накладных расходов, связанных с управлением регистровыми станциями. Однако увеличение пропускной

способности означает, что программа будет выполняться быстрее по сравнению с простой неконвейерной схемой.

В качестве примера рассмотрим неконвейерную машину с пятью этапами выполнения операций, которые имеют длительность 50, 50, 60, 50 и 50 нс соответственно.

Пусть накладные расходы на организацию конвейерной обработки составляют 5 нс. Тогда среднее время выполнения команды в неконвейерной машине будет равно 260 нс.

Если же используется конвейерная организация, длительность такта будет равна длительности самого медленного этапа обработки плюс накладные расходы, т.е. 65 нс. Это время соответствует среднему времени выполнения команды в конвейере. Таким образом, ускорение, полученное в результате конвейеризации, будет равно:

Среднее время выполнения команды в неконвейерном режиме =260
Среднее время выполнения команды в неконвейерном режиме 65=4

Конвейеризация эффективна только тогда, когда загрузка конвейера близка к полной, а скорость подачи новых команд и операндов соответствует максимальной производительности конвейера. Если произойдет задержка, то параллельно будет выполняться меньше операций и суммарная производительность снизится.

Такие задержки могут возникать в результате возникновения конфликтных ситуаций. В следующих разделах будут рассмотрены различные типы конфликтов, возникающие при выполнении команд в конвейере, и способы их разрешения.

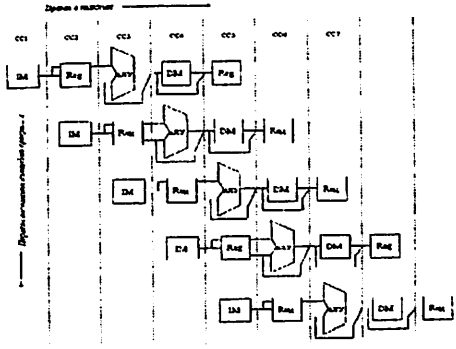


Рис.9 - Представление о работе конвейера

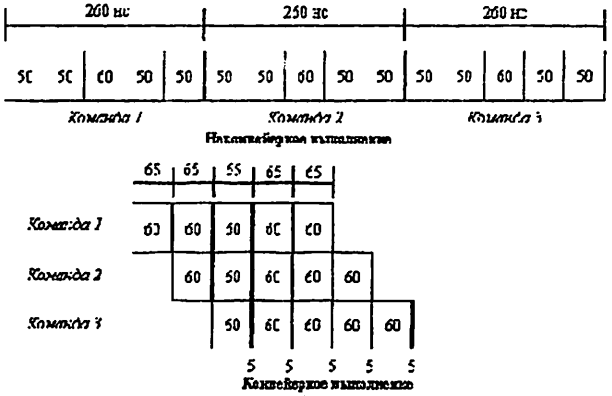


Рис.10 - Эффект конвейеризации при выполнении 3-х команд - четырехкратное ускорение

Структурные конфликты и способы их минимизации

Совмещенный режим выполнения команд в общем случае требует конвейеризации функциональных устройств и дублирования ресурсов для разрешения всех возможных комбинаций команд в конвейере. Если какая-нибудь комбинация команд не может быть принята из-за конфликта по ресурсам, то говорят, что в машине имеется структурный конфликт. Наиболее типичным примером машин, в которых возможно появление структурных конфликтов, являются машины с не полностью конвейерными функциональными устройствами.

Время работы такого устройства может составлять несколько тактов синхронизации конвейера. В этом случае последовательные команды, которые используют данное функциональное устройство, не могут поступать в него в каждом такте.

Другая возможность появления структурных конфликтов связана с недостаточным дублированием некоторых ресурсов, что препятствует выполнению произвольной последовательности команд в конвейере без его приостановки.

Например, машина может иметь только один порт записи в регистровый файл, но при определенных обстоятельствах конвейеру может потребоваться выполнить две записи в регистровый файл в одном такте. Это также приведет к структурному конфликту. Когда последовательность команд наталкивается на такой конфликт, конвейер приостанавливает выполнение одной из команд до тех пор, пока не станет доступным требуемое устройство.

В качестве исторической справки также хотелось бы упомянуть компьютеры фирмы FPS (AP-120B, AP-190L и все более поздние под маркой FPS), также основанные на VLIW-архитектуре, которые были в свое время достаточно распространенными и успешными на рынке. Кроме этого, существовали такие «канонические» машины, как M10 и M13 Карцева, а также «Эльбрус-3» — при всем «неуспехе» последнего проекта, он все же явился этапом VLIW. Вообще, быстродействие VLIW-процессора в большей степени зависит от компилятора, нежели от аппаратуры, поскольку здесь эффект от оптимизации последовательности операций превышает результат, возникающий от повышения частоты.

Относительно недавно мы были свидетелями «противостояния» CISC против RISC, а теперь уже намечается новое «сражение» — VLIW против RISC. Строго говоря, VLIW и суперскалярный RISC — никак не антагонисты, ни в коей мере. Справедливости ради необходимо отметить, что последние — это вовсе не «внешнеархитектурное» свойство, а просто некий способ исполнения.

Возможно, что в дальнейшем появятся суперскалярные VLIW-процессоры, которые тем самым приобретут, если так можно выразиться, «параллелизм в квадрате» — объединение явного статического параллелизма с неявным динамическим. Но на сегодняшнем этапе развития процессоров нет видимых способов совмещать статическое и динамическое переупорядочивание. Именно поэтому Itanium/Itanium2 сейчас следует рассматривать не столько в контексте сравнения VLIW «против» CISC (и тем более не VLIW «против» O3E), а скорее как «синхронный VLIW» vs «асинхронный (Out-Of-Order) RISC». Ну, и не стоит забывать, что альянс Intel-HP придумали для своей архитектуры отдельное название — EPIC, т.е. явный параллелизм.

Несмотря на то, что архитектура VLIW появилась еще на заре компьютерной индустрии (Тьюринг разработал VLIW-компьютер еще в 1946 году), она до сих пор не имела коммерческого успеха. Теперь Intel воплотила некоторые идеи VLIW в линейке процессоров Itanium. Но значительного повышения производительности и скорости вычислений в системах на базе этих процессоров по отношению к существующим классическим «RISC-inside CISC-outside» архитектурам можно добиться лишь путем переноса интеллектуальных функций из аппаратного обеспечения в программное (компилятор). Таким образом, успех Itanium/Itanium2 определяется в основном программными средствами — именно в этом и состоит проблема. Причем довольно сдержанное отношение индустрии к сравнительно давно существующему Itanium только подтвердило факт ее наличия.

EPIC: явный параллелизм команд

Концепция реализации параллелизма на уровне команд (Explicitly Parallel Instruction Computing) определяет новый тип архитектуры, способной конкурировать по масштабам влияния с RISC. Эта идеология направлена на то, чтобы упростить аппаратное обеспечение и, в то же время, извлечь как можно больше «скрытого параллелизма» на уровне команд, используя большую ширину «выдачи» команд (WIW -Wide Issue-Width) и длинные (глубокие) конвейеры с большой задержкой (DPL — Deep Pipeline-Latency), чем это можно сделать при реализации VLIW или суперскалярных стратегий.

EPIC упрощает два ключевых момента, реализуемых во время выполнения. Во-первых, его принципы позволяют во время исполнения отказаться от проверки зависимостей между операциями, которые компилятор уже объявил как независимые.

Во-вторых, данная архитектура позволяет отказаться от сложной логики внеочередного исполнения операций, полагаясь на порядок выдачи команд, определенный компилятором. Более того, EPIC совершенствует возможность компилятора статически генерировать планы выполнения за счет поддержки разного рода перемещений кода во время компиляции, которые были бы некорректными в последовательной архитектуре.

Более ранние решения достигали этой цели главным образом за счет серьезного увеличения сложности аппаратного обеспечения, которое стало настолько значительным, что превратилась в препятствие, не позволяющее отрасли добиваться еще более высокой производительности. EPIC разработан именно для того, чтобы обеспечить более высокую степень параллелизма на уровне команд, поддерживая при этом приемлемую сложность аппаратного обеспечения.

Более высокая производительность достигается как за счет увеличения скорости передачи сигналов, так и благодаря увеличению плотности расположения функциональных устройств на кристалле. Зафиксировав рост этих двух составляющих, дальнейшего увеличения скорости выполнения программ можно добиться в первую очередь благодаря реализации определенного вида параллелизма.

Так, параллелизм на уровне команд (ILP — Instruction-Level Parallelism) стал возможен благодаря созданию процессоров и методик компиляции, которые ускоряют работу за счет параллельного выполнения отдельных RISC-операций.

Системы на базе ILP используют программы, написанные на традиционных языках высокого уровня, для последовательных процессоров, а обнаружение «скрытого параллелизма» автоматически выполняется благодаря применению соответствующей компиляторной технологии и аппаратного обеспечения.

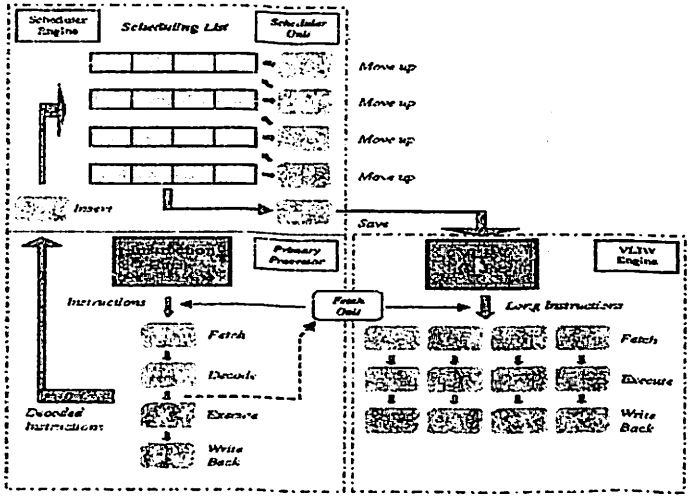


Рис.13 – Процесс работы VLIW архитектуры

Тот факт, что эти методики не требуют от прикладных программистов дополнительных усилий, имеет крайне важное значение, поскольку данное решение резко отличается от традиционного микропроцессорного параллелизма, который предполагает, что программисты должны переписывать свои приложения.

Параллельная обработка на уровне команд является единственным надежным подходом, позволяющим добиться увеличения производительности без фундаментальной переработки приложения.

Суперскалярные процессоры — это реализации ILP-процессора для последовательных архитектур, программа для которых не должна передавать и, фактически, не может передавать точную информацию о параллелизме. Поскольку программа не содержит точной информации о наличии ILP, задача обнаружения параллелизма должна решаться аппаратурой, которая, в свою очередь, должна создавать план действий для обнаружения «скрытого параллелизма».

Процессоры VLIW представляют собой пример архитектуры, для которой программа предоставляет точную информацию о параллелизме — компилятор выявляет параллелизм в программе и сообщает аппаратному обеспечению какие операции не зависят друг от друга. Эта информация имеет важное значение для физического слоя, поскольку в этом случае он «знает» без дальнейших проверок какие операции можно начинать выполнять в одном и том же такте.

Архитектура EPIC — это эволюция архитектуры VLIW, которая абсорбировала в себе многие концепции суперскалярной архитектуры, хотя и в форме, адаптированной к EPIC. По сути — это «идеология», определяющая, как создавать ILP-процессоры, а также набор характеристик архитектуры, которые поддерживают данную основу.

В таком смысле EPIC похож на RISC: определяющий класс архитектур, подчиняющихся общим основным принципам. Точно также, как существует множество различных архитектур наборов команд (ISA) для RISC, может существовать и больше одной ISA для EPIC. В зависимости от того, какие из характеристик EPIC использует архитектура EPIC ISA, она может быть оптимизирована для различных приложений — например, для систем общего назначение или встроенных устройств. Первым примером коммерческой EPIC ISA стала архитектура IA-64.

Код для суперскалярных процессоров содержит последовательность команд, которая порождает корректный результат, если выполняется в установленном порядке.

Код указывает последовательный алгоритм и, за исключением того, что он использует конкретный набор команд, не представляет себе точно природу аппаратного обеспечения, на котором он будет работать или точный временной порядок, в котором будут выполняться команды.

В отличие от программ для суперскалярных процессоров, код VLIW предлагает точный план (POE — Plan Of Execution, схема исполнения создается статически во время компиляции) того, как процессор будет выполнять программу. Код точно указывает когда будет выполнена каждая операция, какие функциональные устройства будут работать и какие регистры будут содержать операнды.

Компилятор VLIW создает такой план выполнения, имея полное представление о самом процессоре, чтобы добиться требуемой записи исполнения (ROE — Record Of Execution) — последовательности событий, которые действительно происходят во время работы программы. Компилятор передает POE (через архитектуру набора команд, которая точно описывает параллелизм) аппаратному обеспечению, которое, в свою очередь, выполняет указанный план.

Этот план позволяет VLIW использовать относительно простое аппаратное обеспечение, способное добиться высокого уровня ILP. В отличие от VLIW, суперскалярная аппаратура динамически строит POE на основе последовательного кода. Хотя такой подход и увеличивает сложность физической реализации, суперскалярный процессор создает план, используя преимущества тех факторов, которые могут быть определены только во время выполнения.

Одна из целей, которые ставили перед собой при создании EPIC, состояла в том, чтобы сохранить реализованный во VLIW принцип статического создания POE, но и в то же время обогатить его возможностями, аналогичными возможностям супер скалярного процессора, позволяющими новой архитектуре лучше учитывать динамические факторы, традиционно ограничивающие параллелизм, свойственный VLIW. Чтобы добиться этих целей, «идеология» EPIC была построена на некоторых основных принципах.

Первый — это создание плана выполнения во время компиляции. EPIC возлагает нагрузку по созданию POE на компилятор. Хотя, в общем, архитектура и физическая реализация могут препятствовать компилятору в выполнении этой задачи, процессоры EPIC предоставляют функции, которые помогают компилятору создавать план выполнения.

Во время исполнения поведение процессора EPIC с точки зрения компилятора должно быть предсказуемым и управляемым. Динамическое внеочередное исполнение команд может «запутать» компилятор так, что он не будет «понимать», как его решения повлияют на реальную запись выполнения, созданную процессором, поэтому ему необходимо уметь предсказывать действия процессора, что еще больше усложняет задачу.

В данной ситуации предпочтителен процессор, четко исполняющий то, что ему указывает программа. Суть же создания плана во время компиляции состоит в переупорядочивании исходного последовательного кода так, чтобы использовать все преимущества параллелизма приложения и максимально эффективно тратить аппаратные ресурсы, минимизируя время выполнения. Без соответствующей поддержки архитектуры такое переупорядочивание может нарушить корректность программы.

Таким образом, поскольку EPIC возлагает создание POE на компилятор, она должна обеспечивать еще и архитектурные возможности, поддерживающие интенсивное переупорядочивание кода во время компиляции.

Следующим принципом является использование компилятором вероятностных оценок. Компилятор EPIC сталкивается с серьезной проблемой при создании плана выполнения: информация определенного типа, которая существенно влияет на запись исполнения, становится известна только лишь в момент выполнения программы.

Например, компилятор не может точно знать какая из ветвей после оператора перехода будет выполняться, когда запланированный код пройдет базовые блоки и какой из путей графа будет выбран. Кроме того, обычно невозможно создать статический план, который одновременно оптимизирует все пути в программе.

Неоднозначность также возникает и в тех случаях, когда компилятор не может решить, будут ли ссылки указывать на одно и то же место в памяти. Если да, то обращение к ним должно осуществляться последовательно; если нет, то их можно запланировать в произвольном порядке.

При такой неоднозначности часто наиболее вероятен некий конкретный результат. Одним из важнейших принципов EPIC в данной ситуации является возможность разрешения компилятору оперировать вероятностными оценками — он создает и оптимизирует POE для наиболее вероятных случаев. Однако EPIC обеспечивает архитектурную поддержку, такую как спекулятивное выполнение по управлению и по данным (Control and Data Speculation), с тем, чтобы гарантировать корректность программы, даже если исходные предположения были не верны. Когда предположение оказывается неверным, возвращено очевидно падение производительности при выполнении программы.

Такой эффект производительности иногда виден на плане программы, к примеру, в тех случаях, когда существует высоко оптимизированная программная область, а код исполняется в менее оптимизированной. Также падение производительности может возникнуть в моменты «остановки» (Stall), которые на плане программы не видны — определенные операции, подпадающие под наиболее вероятный и, следовательно, оптимизированный случай, выполняются при максимальной производительности, но приостанавливают процессор для того, чтобы гарантировать корректность, если возникнет менее вероятный, не оптимизированный случай.

После того, как создан план, компилятор передает его аппаратному обеспечению. Для этого ISA должен обладать возможностями достаточно богатыми, чтобы сообщить решения компилятора о том, когда инициировать каждую операцию и какие ресурсы использовать (в частности, должен существовать способ указать, какие операции инициируются одновременно).

В качестве альтернативного решения компилятор мог бы создавать последовательную программу, которую процессор динамически реорганизует с тем, чтобы получить требуемую запись. Но в таком случае цель, сводимая к освобождению аппаратного обеспечения от динамического планирования, не достигается. При передаче POE аппаратному обеспечению крайне важно своевременно предоставить необходимую информацию.

Примером этому может служить операция перехода, которая, в случае ее использования, требует, чтобы по адресу перехода команды выбирались с упреждением, заведомо до того, как будет инициирован сам переход. Вместо того, чтобы решение о том, когда это нужно сделать и какой адрес перехода отдавать на откуп аппаратному обеспечению, такая информация в соответствии с основными принципами EPIC передается аппаратному обеспечению точно и своевременно через код.

Микроархитектура принимает и другие решения, не связанные напрямую с выполнением кода, но которые влияют на время выполнения. Один из таких примеров — управление иерархией кэш-памяти и соответствующие решения о том, какие данные нужны для поддержки иерархии, а какие следует заменить.

Такие правила обычно предусматриваются алгоритмом функционирования контроллера кэша. EPIC расширяет принцип, утверждающий, что план выполнения компилятор создает так, чтобы тоже иметь возможность управлять этими механизмами микроархитектуры.

Для этого обеспечиваются архитектурные возможности, позволяющие осуществлять программный контроль механизмами, которыми обычно управляет микроархитектура.

Аппаратно-программный комплекс VLIW

Архитектура VLIW представляет собой одну из реализаций концепции внутреннего параллелизма в микропроцессорах. Их быстродействие можно повысить двумя способами: увеличив либо тактовую частоту, либо количество операций, выполняемых за один такт.

В первом случае требуется применение «быстрых» технологий (например, использование арсенида галлия вместо кремния) и таких архитектурных решений, как глубинная конвейеризация (конвейеризация в пределах одного такта, когда в каждый момент времени задействованы все логические блоки кристалла, а не отдельные его части). Для увеличения количества выполняемых за один цикл операций необходимо на одном чипе разместить множество функциональных модулей обработки и обеспечить надежное параллельное исполнение машинных инструкций, что дает возможность включить в работу все модули одновременно.

Надежность в таком контексте означает, что результаты вычислений будут правильными. Для примера рассмотрим два выражения, которые связаны друг с другом следующим образом: A=B+C и B=D+E. Значение переменной A будет разным в зависимости от порядка, в котором вычисляются эти выражения (сначала A, а потом B, или наоборот), но ведь в программе подразумевается только одно определенное значение. И если теперь вычислить эти выражения параллельно, то на правильный результат можно рассчитывать лишь с определенной вероятностью, а не гарантировано.

Планирование порядка вычислений — довольно трудная задача, которую приходится решать при проектировании современного процессора. В суперскалярных архитектурах для распознавания зависимостей между машинными инструкциями применяется специальное довольно сложное аппаратное решение (например, в P6- и post-P6-архитектуре от Intel для этого

используется буфер переупорядочивания инструкций — ReOrder Buffer, ROB).

Однако размеры такого аппаратного планировщика при увеличении количества функциональных модулей обработки возрастают в геометрической прогрессии, что, в конце концов, может занять весь кристалл процессора. Поэтому суперскалярные проекты остановились на отметке 5-6 обрабатываемых за цикл инструкций. На самом же деле, текущие реализации VLIW тоже далеко не всегда могут похвастаться 100% заполнением пакетов — реальная загрузка около 6-7 команд в такте — примерно столько же, сколько и лидеры среди RISC-процессоров.

При другом подходе можно передать все планирование программному обеспечению, как это делается в конструкциях с VLIW. «Умный» компилятор должен выискать в программе все инструкции, которые являются совершенно независимыми, собрать их вместе в очень длинные строки (длинные инструкции) и затем отправить на одновременное исполнение функциональными модулями, количество которых, как минимум, не меньше, чем количество операций в такой длинной команде.

Очень длинные инструкции (VLIW) обычно имеют размер 256-1024 bit, однако, бывает и меньше. Сам же размер полей, кодирующих операции для каждого функционального модуля, в такой метаинструкции намного меньше.

Логический слой VLIW-процессора

Процессор VLIW, имеющий схему, представленную ниже, может выполнять в предельном случае восемь операций за один такт и работать при меньшей тактовой частоте намного более эффективнее существующих суперскалярных чипов.

Добавочные функциональные блоки могут повысить производительность (за счет уменьшения конфликтов при распределении ресурсов), не слишком усложняя чип.

Однако такое расширение ограничивается физическими возможностями: количеством портов чтения/записи, необходимых для обеспечения одновременного доступа функциональных блоков к файлу регистров, и взаимосвязей, число которых геометрически растет при увеличении количества функциональных блоков. К тому же компилятор должен распараллелить программу до необходимого уровня, чтобы обеспечить загрузку каждому блоку — это, думается, самый главный момент, ограничивающий применимость данной архитектуры.

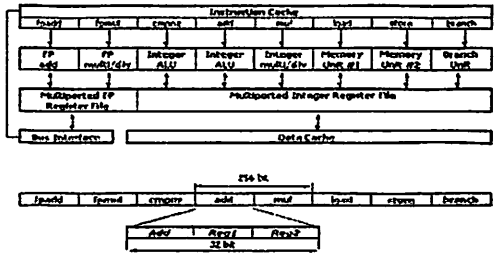


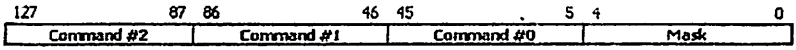
Рис.14 – Логический слой VLIW архитектуры

Эта гипотетическая инструкция имеет восемь операционных полей, каждое из которых выполняет традиционную трехоперандную RISC-подобную инструкцию типа <регистр приемника> = <регистр источника 1> — <операция> — <регистр источника 2> (типа классической команды MOV AX BX) и может непосредственно управлять специфическим функциональным блоком при минимальном декодировании.

Для большей конкретности рассмотрим кратко IA-64 как один из примеров воплощения VLIW. Со временем эта архитектура способна вытеснить x86 (IA-32) не только на рынке, но вообще как класс, хотя это уже удел далекого будущего. Тем не менее, необходимость разработки для IA-64 весьма сложных компиляторов и трудности с созданием оптимизированных машинных кодов может вызвать дефицит специалистов, работающих на ассемблере IA-64, особенно на начальных этапах, как самых сложных.

Наиболее кардинальным нововведением IA-64 по сравнению с RISC является явный параллелизм команд (EPIC), вносящий некоторые элементы, напоминающие архитектуру сверхдлинного командного слова, которые назвали связками (bundle). Так, в обеих архитектурах явный параллелизм представлен уже на уровне команд, управляющих одновременной работой функциональных исполнительных устройств (или функциональных модулей, или просто функциональных устройств, ФУ).

В данном случае связка имеет длину 128bit и включает в себя 3 поля для команд длиной 41bit каждое, и 5-разрядный слот шаблона. Предполагается, что команды связки могут выполняться параллельно разными ФУ. Возможные взаимозависимости, препятствующие параллельному выполнению команд одной связки, отражаются в поле шаблона. Не утверждается, впрочем, что параллельно не могут выполняться и команды разных связок. Хотя, на основании заявленного уровня параллельности исполнения, достигающего шести команд за такт, логично предположить, что одновременно могут выполняться как минимум две связки.



Шаблон указывает какого непосредственно типа команды находятся в слотах связки. В общем случае однотипные команды могут выполняться в более чем одном типе функциональных устройств. Шаблоном задаются так называемые остановки, определяющие слот, после начала выполнения команд которого инструкции последующих полей должны ждать завершения. Порядок слотов в связке (более важные справа) отвечает и порядку байт (Little Endian), однако данные в памяти могут располагаться и в режиме Big Endian (более важные слева), который устанавливается специальным битом в регистре маски пользователя.

Вращение регистров является в некотором роде частным случаем переименования регистров, применяемого во многих современных суперскалярных процессорах с внеочередным спекулятивным («умозрительным») выполнением команд. В отличие от них, вращение регистров в IA-64 управляется программно. Использование этого механизма в IA-64 позволяет избежать накладных расходов, связанных с сохранением/восстановлением большого числа регистров при вызовах подпрограмм и возвратах из них, однако статические регистры при необходимости все-таки приходится сохранять и восстанавливать, явно кодируя соответствующие команды.

К слову, система команд IA-64 довольно уникальна. Среди принципиальных особенностей следует отдельно отметить спекулятивное выполнение команд и применение предикатов — именно это подмножество и определяет исключительность IA-64. Все подобные команды можно подразделить на команды работы со стеком регистров, целочисленные команды, команды сравнения и работы с предикатами, команды доступа в память, команды перехода, мультимедийные команды, команды пересылок между регистрами, «разные» (операции над строками и подсчет числа единиц в слове) и команды работы с плавающей запятой.

Аппаратная реализация VLIW-процессора очень проста: несколько небольших функциональных модулей (сложения, умножения, ветвления и т.д.), подключенных к шине процессора, и несколько регистров и блоков кэш-памяти. VLIW-архитектура представляет интерес для полупроводниковой промышленности по двум причинам.

Первая — теперь на кристалле больше места может быть отведено для блоков обработки, а не, скажем, для блока предсказания переходов. Вторая причина — VLIW-процессор может быть высокоскоростным, так как предельная скорость обработки определяется только внутренними особенностями самих функциональных модулей. Привлекает и то, что VLIW при определенных условиях может реализовать старые CISC-инструкции эффективнее RISC. Это потому, что программирование VLIW-процессора

очень напоминает написание микрокода (исключительно низкоуровневый язык, позволяющий всесторонне программировать физический слой, синхронизируя работу логических вентилей с шинами обмена данными и управляя передачей информации между функциональными модулями).

В те времена, когда память для ПК была дорогостоящей, программисты экономили ее, прибегая к сложным инструкциям процессора x86 типа STOS и LODS (косвенная запись/чтение в/из памяти). CISC реализует такие инструкции, как микропрограммы, зашитые в постоянную память (ROM) и выполняемые процессором.

Архитектура RISC вообще исключает использование микрокода, реализуя инструкции чисто аппаратным путем — фактически, инструкции RISC-процессора почти аналогичны микрокоду, используемому в CISC. VLIW делает по-другому — изымает процедуру генерирования микрокода из процессора (да и вообще стадии исполнения) и переносит его в компилятор, на этап создания исполняемого кода. В результате эмуляция инструкций процессора x86, таких как STOS, осуществляется очень эффективно, поскольку процессор получает для исполнения уже готовые макросы.

Но вместе с тем, это порождает и некоторые трудности, поскольку написание достаточно эффективного микрокода — невероятно трудоемкий процесс. Архитектуре VLIW может обеспечить жизнеспособность только «умный» компилятор, который возьмет эту работу на себя. Именно это обстоятельство ограничивает использование вычислительных машин с архитектурой VLIW: пока они нашли свое применение в основном в векторных (для научных расчетов) и сигнальных процессорах.

VLIW: обратная сторона медали

Тем не менее, при реализации архитектуры VLIW возникают и другие серьезные проблемы. VLIW-компилятор должен в деталях «знать» внутренние особенности архитектуры процессора, опускаясь до устройства самих функциональных блоков.

Как следствие, при выпуске новой версии VLIW-процессора с большим количеством обрабатывающих модулей (или даже с тем же количеством, но другим быстродействием) все старое программное обеспечение может потребовать полной перекомпиляции. Производители VLIW-процессоров обрекли себя на, как минимум, не уменьшение ширины пакета, хотя бы ради того, чтобы «старые» программы могли гарантированно исполнятся на новых устройствах.

Например, на настоящее время существуют пакеты по 8 команд, а следующая версия не может иметь в реализации всего шесть функциональных устройств даже ради двух-трех- кратного прироста по частоте. Кроме того, что программа, скомпилированная для восьмиканального VLIW, не сможет без специальных дорогостоящих (и в

плане сложности, и в плане производительности) аппаратных решений исполняться на шестиканальной архитектуре, придется радикально переписывать и компилятор. С этой точки зрения представляется разумным использование

Intel трехкомандного слова в системе IA64 — такое, на первый взгляд, неудобное ограничение позволяет в будущем довольно свободно варьировать число исполнительных устройств в процессорах IA64. И если при переходе с 386 на процессор 486 производить перекомпиляцию имеющегося ПО было совершенно ненужно, то теперь придется.

В качестве одного из возможных компромиссных решений предлагается разделить процесс компиляции на две стадии. Все программное обеспечение должно готовиться в аппаратно-независимом формате с использованием промежуточного кода, который окончательно транслируется в машинно-зависимый код только в процессе установки на оборудовании конечного пользователя.

Пример такого подхода демонстрирует фонд OSF со своим архитектурно-независимым форматом ANDF (Architecture-Neutral Distribution Format). Но кроссплатформенное программное обеспечение пока что не оправдывает когда-то возлагавшихся на него радужных надежд.

Во-первых, оно все равно требует наличия портов, которые «объясняют» компилятору в каждом конкретном случае что следует делать при компиляции данной программы именно на этой платформе (и даже именно на этой ОС).

Во-вторых, кроссплатформенное ПО пока отнюдь не является «Speed Demon», а даже наоборот, как правило работает медленнее, чем написанное под конкретную платформу аналогичного класса приложения.

Другая трудность — это по своей сути статическая природа оптимизации, которую обеспечивает VLIW-компилятор. Трудно предугадать как, например, поведет себя программа, когда столкнется во время компиляции с непредусмотренными динамическими ситуациями, такими как ожидание ввода/вывода.

Архитектура VLIW возникла в ответ на требования со стороны научно-технических организаций, где при вычислениях особенно необходимо большое быстродействие процессора, но для объектно-ориентированных и управляемых по событиям программ она менее подходит, а ведь именно такие приложения составляют сейчас большинство в сфере Информационных Технологий.

Остается труднопредполагаемой проверка, что компилятор выполняет такие сложные преобразования надежно и правильно. Напротив, Out-Of-Order RISC-процессоры вполне способны «адаптироваться» под конкретную ситуацию самым выгодным образом.

Сейчас уже стало очевидным, что разработчики современных быстродействующих VLIW-процессоров начинают отходить от своей первоначальной затеи с чистой VLIW-архитектурой. Intel, по крайней мере, сохранила семейству Itanium возможность исполнять классический x86-код, правда, судя по всему, удалось ей это сделать ценой колоссальных потерь производительности при работе процессора в таком режиме. Можно предположить, что процессорный гигант сдал бастион «чистого VLIW» под нажимом гигантов ПО.

Однако решение сложной задачи обеспечения взаимодействия аппаратного и программного обеспечения в архитектуре VLIW требует серьезных предварительных исследований. На данный момент теоретические изыскания в области алгоритмов автоматического распараллеливания программ все еще не могут похвастаться большим количеством достижений: как правило, если требуется грамотно распараллелить программу (к примеру, для введения в нее SMP-оптимизированных участков), делается это по старинке — «руками», причем руками весьма и весьма квалифицированных программистов. К тому же давно известен немалый список задач, поддающихся распараллеливанию с громадными усилиями или даже не поддающихся вообще.

Как пример, можно вспомнить знаменитый Crusoe от Transmeta. Принцип его работы фактически состоит в динамической перекомпиляции под VLIW-архитектуру уже скомпилированного x86 кода. Однако, если даже в создании эффективных VLIW-компиляторов для языков высокого уровня разработчики сталкиваются с такими большими сложностями, то что можно сказать о «компиляторе» весьма хаотичного и непредсказуемого машинного кода x86, тем более, что он зачастую опять-таки оптимизирован при компиляции, но в расчете на совсем другие архитектурные особенности.

Реально же Crusoe применяется только в режиме эмуляции x86, хотя принципиальных ограничений на эмуляцию любого другого кода нет. При этом все программы и сама операционная система работают поверх низкоуровневого программного обеспечения, называемого морфингом кода (Code Morphing) и ответственного за трансляцию x86-кодов в связки, имеющие размер 128bit. Crusoe использует также собственную терминологию, тонко определяющую уровень логического слоя архитектуры, в которой связки называются молекулами (Molecule), а 32bit подкоманды, располагаемые в связке, — атомами (Atom).

Каждая молекула содержит два или четыре атома. Для совместимости с форматом команд, если при двоичной трансляции не удается заполнить все «атомные» слоты молекулы, в незаполненные поля должна вставляется пустая подкоманда NOP, указывающая на отсутствие операции (No

OPeration). В результате, благодаря применению двух типов молекул, в каждой из них оказывается не более одной NOP.

Всего же за такт могут исполняться до четырех подкоманд VLIW. Среди важных архитектурных особенностей VLIW-ядра Crusoe выделяются относительно короткие конвейеры: целочисленный на семь стадий и на десять с плавающей запятой.

Теоретически данный процессор может применяться для эмуляции различных архитектур, однако ряд особенностей его микроархитектуры нацелен на эффективную эмуляцию именно x86 кода. Принципиально важным отличием является практически отсутствие потерь производительности в условиях примерно равной частоты процессоров при x86-эмуляции. Сначала устройство декодирует x86-последовательность в режиме интерпретации «байт за байтом», однако если код выполняется несколько раз, то механизм морфинга транслирует его в оптимальную последовательность молекул, а результат трансляции кэшируется для повторного использования.

Можно сказать, что Transmeta попыталась не то, что опередить время, а даже «перескочить через голову» уже и так опередившей время технологии. В принципе же то, что сделала Transmeta — просто фантастическое техническое достижение.

По сути дела, продемонстрирована работающая в реальном времени технология динамической компиляции кроссплатформенного ПО. Если так работает система, переводящая «на лету» x86 код во внутреннее представление, то остается только догадываться как бы она работала с первоначально ориентированной для нее программой. Таким образом, бинарная совместимость возможна и довольно эффективна.

С другой стороны, даже сам Гордон Мур уже заявил, что рост частоты процессоров в ближайшее время скорее всего перестанет подчиняться сформулированному им эмпирическому правилу, и существенно замедлится. В этой ситуации производительность классических систем рискует довольно скоро упереться в «тупик» физических ограничений, что в еще большей степени подхлестнет развитие альтернативных повышению частоты способов увеличения быстродействия, одним из которых является разработка принципиально новых и хорошо распараллеливающихся алгоритмов, что и позволит проявить себя VLIW во всей красе.

Контрольные вопросы:

1.Для чего применялись процессоры с архитектурой VLIW на заре эволюции суперкомпьютеров?

2.Расскажите про первый мини компьютер с процессором на архитектуре VLIW.
3.EPIC.Преимущества и недостатки перед RISC архитектурой.
4.Нужна ли VLIW архитектура, когда есть RISC или CISC архитектуры? Объясните свой ответ, приведя преимущества и недостатки.
5.Какая концепция используется в VLIW архитектуре?

Содержание отчёта

1. Использовать листы формата А4
2. Использовать шрифта Times New Roman, размер кегля 14
3. Интервал 1,5
4. Написать тему, цель и задачи практического работы
5. Выполнить задание
6. Написать выводы и ответить письменно на вопросы.

Практическое занятие №6

Тема: Изучение модели вычислений "Операция-операнд"
Цель работы: Изучение модели вычислений "Операция-операнд".

Теоретическая часть

При разработке параллельных алгоритмов решения сложных научно-технических задач принципиальным моментом является анализ эффективности использования параллелизма, состоящий обычно в оценке получаемого ускорения процесса вычислений (сокращения времени решения задачи).

Формирование подобных оценок ускорения может осуществляться применительно к выбранному вычислительному алгоритму (оценка эффективности распараллеливания конкретного алгоритма). Другой важный подход может состоять в построении оценок максимально возможного ускорения процесса решения задачи конкретного типа (оценка эффективности параллельного способа решения задачи).

В данной работе описывается модель вычислений в виде графа "операцииоперанды", которая может использоваться для описания существующих информационных зависимостей в выбираемых алгоритмах решения задач, приводятся оценки эффективности максимально возможного параллелизма, которые могут быть получены в результате анализа имеющихся моделей вычислений.

Для описания существующих информационных зависимостей в выбираемых алгоритмах решения задач может быть использована модель в виде графа «операции-операнды».

Принимается:

- время выполнения любых вычислительных операций является одинаковым и равняется 1 (в тех или иных единицах измерения).
- передача данных между вычислительными устройствами выполняется мгновенно без каких-либо затрат времени (что может быть справедливо, например, при наличии общей разделяемой памяти в параллельной вычислительной системе).

Модель вычислений в виде графа "операции-операнды". Для описания существующих информационных зависимостей в выбираемых алгоритмах решения задач может быть использована модель в виде графа "операциионеранды". Для уменьшения сложности излагаемого материала при построении модели будет предполагаться, что время выполнения любых вычислительных операций является одинаковым и равняется 1 (в тех или иных единицах измерения). Кроме того, принимается, что передача данных между вычислительными устройствами выполняется мгновенно без каких-либо затрат времени (что может быть справедливо, например, при наличии общей разделяемой памяти в параллельной вычислительной системе).

Представим множество операций, выполняемых в исследуемом алгоритме решения вычислительной задачи, и существующие между операциями информационные зависимости в виде ациклического ориентированного графа

$$G = (V, R),$$

где $V = \{1,...,V\}$ есть множество вершин графа, представляющих выполняемые операции алгоритма, а R есть множество дуг графа (при этом дуга $r = (i, j)$ принадлежит графу только, если операция j использует результат выполнения операции i). Для примера на рисунке ниже показан граф алгоритма вычисления площади прямоугольника, заданного координатами двух противолежащих углов. Как можно заметить по приведенному примеру, для выполнения выбранного алгоритма решения задачи могут быть использованы разные схемы вычислений и построены соответственно разные вычислительные модели. Как будет показано далее, разные схемы вычислений обладают различными возможностями для распараллеливания и, тем самым, при построении модели вычислений может быть поставлена задача выбора наиболее подходящей для параллельного исполнения вычислительной схемы алгоритма. В рассматриваемой

вычислительной модели алгоритма вершины без входных дуг могут использоваться для задания операций ввода, а вершины без выходных дуг – для операций вывода. Обозначим через V множество вершин графа без вершин ввода, а через $d(G)$ диаметр (длину максимального пути) графа.

Описание схемы параллельного выполнения алгоритма Операции алгоритма, между которыми нет пути в рамках выбранной схемы вычислений, могут быть выполнены параллельно (для вычислительной схемы на рис. 2.1, например, параллельно могут быть реализованы сначала все операции умножения, а затем первые две операции вычитания). Возможный способ описания параллельного выполнения алгоритма может состоять в следующем.

Пусть p есть количество процессоров, используемых для выполнения алгоритма. Тогда для параллельного выполнения вычислений необходимо задать множество (расписание)

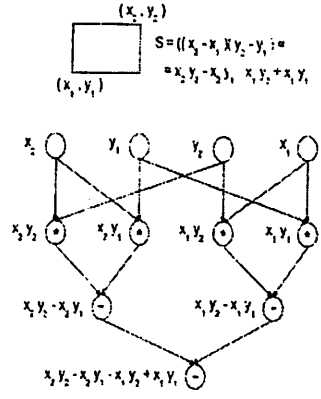


Рис. 15 – Операции алгоритма

Пример вычислительной модели алгоритма в виде графа "операциионеранды" в котором для каждой операции указывается номер используемого для выполнения операции процессора P_i и время начала выполнения операции t_i. Для того, чтобы расписание было реализуемым, необходимо выполнение следующих требований при задании множества H_p:

1) т.е. один и тот же процессор не должен назначаться разным операциям в один и тот же момент времени,

2) $\forall(i,j) \in R \Rightarrow t_j \geq t_i + 1$, т.е. к назначаемому моменту выполнения операции все необходимые данные уже должны быть вычислены.

Определение времени выполнения параллельного алгоритма. Вычислительная схема алгоритма G совместно с расписанием H p может рассматриваться как модель параллельного алгоритма $A_p(G,H_p)$, исполняемого с использованием p процессоров. Время выполнения параллельного алгоритма определяется максимальным значением времени, используемым в расписании

$$T_p(G,H_p) = \max_{i \in V}(t_i + 1).$$

Для выбранной схемы вычислений желательно использование расписания, обеспечивающего минимальное время исполнения алгоритма

$$T_p(G) = \min_{H_p} T_p(G,H_p).$$

Уменьшение времени выполнения может быть обеспечено и путем подбора наилучшей вычислительной схемы

$$T_p = \min_G T_p(G).$$

Оценки $T_p(G, H_p)$, $T(G)$ и T_p могут быть использованы в качестве показателей времени выполнения параллельного алгоритма. Кроме того, для анализа максимально возможного параллелизма можно определить оценку наиболее быстрого исполнения алгоритма:

$$T_\infty = \min_{p \geq 1} T_p$$

Оценку T_∞ можно рассматривать как минимально возможное время выполнения параллельного алгоритма при использовании неограниченного количества процессоров (концепция вычислительной системы с бесконечным количеством процессоров, обычно называемой *паракомпьютером*, широко используется при теоретическом анализе параллельных вычислений).

Оценка T_1 определяет время выполнения алгоритма при использовании одного процессора и представляет, тем самым, время выполнения последовательного варианта алгоритма решения задачи. Построение подобной оценки является важной задачей при анализе параллельных алгоритмов, поскольку она необходима для определения эффекта

использования параллелизма (ускорения времени решения задачи). Очевидно, что:

$$T_1(G) = |\bar{V}|,$$

где $|\bar{V}|$, напомним, есть количество вершин вычислительной схемы G без вершин ввода. Важно отметить, что если при определении оценки T_1 ограничиться рассмотрением только одного выбранного алгоритма решения задачи и использовать величину:

$$T_1 = \min_G T_1\ (G),$$

то получаемые при использовании такой оценки показатели ускорения будут характеризовать эффективность распараллеливания выбранного алгоритма. Для оценки эффективности параллельного решения исследуемой задачи вычислительной математики время последовательного решения следует определять с учетом различных последовательных алгоритмов, т.е. использовать величину:

$$T_1^* = \min T_1,$$

где операция минимума берется по множеству всех возможных последовательных алгоритмов решения данной задачи.

Контрольные вопросы:

1. Что вы знаете о параллельных алгоритмов?
2. Для чего мы используем модель вычисление операция и операнд?
3. Расскажите о преимуществах данной модели.
4. Как определить время выполнения параллельного алгоритма?
5. Подведя итоги формул вычисления модели «операция и операнд», расскажите о недостатках данного метода.

Содержание отчёта

1. Использовать листы формата А4
2. Использовать шрифта Times New Roman, размер кегля 14
3. Интервал 1,5
4. Написать тему, цель и задачи практического работы

5. Выполнить задание
6. Написать выводы и ответить письменно на вопросы.

Практическое занятие № 7

Тема: Изучение работы процессора с виртуальным ядром(NT)

Цели работы: Изучение работы процессора с виртуальным ядром(NT) и его свойств.

Теоретическая часть

Виртуальное ядро процессора — что это?

Дело в том, что процессор работает неидеально. Но сделать идеально — значит начинать все заново, не просто выпустить новый процессор, а вообще все заново, то что начинали лет 20-30 назад.

Поэтому создают технологии, которые ускоряют работу процессора. Одна из них — виртуальные ядра, они же потоки и они же технология Hyper-Threading (HT), это у Intel, а у AMD технология называется SMT (от англ. simultaneous multithreading).

Именно эта технология делает так, что одно физическое ядро представляется в системе как два виртуальных или два потока. Таких два потока — быстрее одного ядра, но медленнее двух настоящих ядер.

Поддержка виртуальных ядер определяется процессором, не все модели эту технологию поддерживают.

Виртуальное ядро процессора — как работает?

На самом деле все просто. Процессор работает себе, трудится, все хорошо. Но в работе могут быть паузы, например:

1. Произошел промах при запросе к кэшу.
2. Неверное предсказание события.
3. Ожидание результата предыдущей инструкции.

Это примерные причины, по которым процессор может простаивать доли секунды. Даже доли секунды в процессорном времени — это значительно, особенно если таких простоев много. Что делать? Все просто. Изначально все что выше написано — делается на одном ядре, а точнее на одном потоке. Но при наличии технологии виртуальных ядер — процессор не будет останавливаться, а просто в это время передаст управление другому потоку, который тоже выполняет определенную работу.

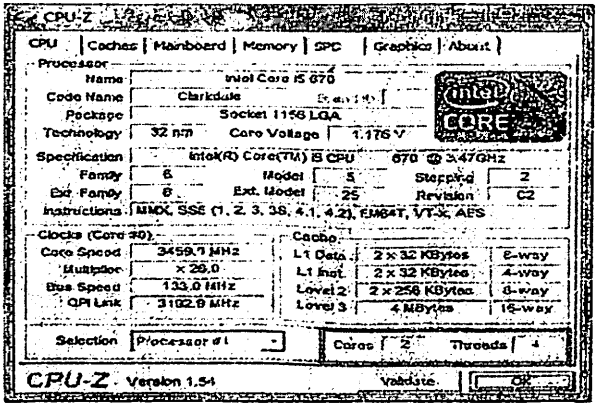


Рис. 17 – утилита CPU-Z

Контрольные вопросы:

1.Для чего нужны ядра процессора?
2.Лучше ли использовать только лишь ядра в использовании компьютера? Объясните свой ответ.
3.Приведите пару примеров процессоров с технологией HT у Intel,а так же пару примеров процессоров с SMT у AMD.
4.К каким последствиям приводит неисправность потоков у процессора?
5.Что такое TDP процессора и как на него влияет количество потоков\ядер?

Содержание отчёта

1. Использовать листы формата А4
2. Использовать шрифта Times New Roman, размер кегля 14
3. Интервал 1,5
4. Написать тему, цель и задачи практического работы
5. Выполнить задание
6. Написать выводы и ответить письменно на вопросы.

Практическое занятие №8

Тема: Алгоритмы работы нейрокомпьютеров

Цель работы: изучить структуру нейрокомпьютера и его основных блоков.

Теоретическая часть

В последнее время активно ведутся также работы по построению моделей обработки информации в нервной системе. Большинство моделей основывается на схеме формального нейрона У Мак Каллока и У.Питтса, согласно которой нейрон представляет собой пороговый элемент, на входах которого имеются возбуждающие и тормозящие синапсы; в этом нейроне определяется взвешенная сумма входных сигналов (с учетом весов синапсов), а при превышении этой суммой порога нейрона вырабатывается выходной сигнал.

В моделях уже построены нейронные сети, выполняющие различные алгоритмы обработки информации: ассоциативная память, категоризация (разбиение множества образов на кластеры, состоящие из подобных друг другу образов), топологически корректное отображение одного пространства переменных в другое, распознавание зрительных образов, инвариантное относительно деформаций и сдвигов в пространстве решение задач комбинаторной оптимизации. Подавляющее число работ относится к исследованию алгоритмов нейросетей с прагматическими целями.

Предполагается, что практические задачи будут решаться нейрокомпьютерами искусственными нейро-подобными сетями, созданными на основе микроэлектронных вычислительных систем. Спектр задач для разрабатываемых нейрокомпьютеров достаточно широк: распознавание зрительных и звуковых образов, создание экспертных систем и их аналогов, управление роботами, создание нейро-протезов для людей, потерявших слух или зрение.

Достоинства нейрокомпьютеров параллельная обработка информации и способность к обучению. Несмотря на чрезвычайную активность исследований по нейронным сетям и нейркомпьютерам, многое в этих исследованиях настораживает.

Ведь изучаемые алгоритмы выглядят как бы "вырванным куском" из общего осмысления работы нервной системы. Часто исследуются те алгоритмы, для которых удается построить хорошие модели, а не те, что наиболее важны для понимания свойств мышления, работы мозга и для создания систем искусственного интеллекта. Задачи, решаемые этими алгоритмами, оторваны от эволюционного контекста, в них практически не

рассматривается, каким образом и почему возникли те или иные системы обработки информации.

Настораживает также чрезмерная упрощенность понимания работы нейронных сетей, при котором нейроны осмыслены лишь как суммирующие пороговые элементы, а обучение сети происходит путем модификации синапсов.

Ряд исследователей, правда, рассматривает нейрон как значительно более сложную систему обработки информации, предполагая, что основную роль в обучении играют молекулярные механизмы внутри нейрона. Все это указывает на необходимость максимально полного понимания работы биологических систем обработки информации и свойств организмов, обеспечиваемых этими системами.

Одним из важных направлений исследований, способствующих такому пониманию, , наверное, может быть анализ того, как в процессе биологической эволюции возникали "интеллектуальные" свойства биологических организмов.

Нейрокомпьютер объект, над созданием которого десятки лет работает огромная армия ученых, технологов, инженеров, математиков всего мира. На эту научную проблему тратятся огромные ресурсы. Только в Японии на восьмилетнюю программу «Нейрокомпьютер» по разработке современного компьютера шестого поколения на 19891997 годы было выделено 231 млн. долларов. Для создания нейрокомпьютера объединились крупнейшие фирмы Японии: Фудзицу, Хитачи, Тошиба, Мицубиси денки, Нихон денки, Оки денки.

Эта задача грандиозна и сложна, в ней переплетены многие фундаментальные проблемы: мозг, психика, квантовая физика, информатика, биоэнергетика, общая теория поля, философия и др. Об актуальности и грандиозности проблемы можно судить по программы Пентагона «Стратегическая компьютерная инициатива», цель которой создание нового поколения компьютеров, обладающих определенными человеческими качествами: «здравым смыслом», специальными знаниями, умением видеть, слышать и говорить.

Попытки создать искусственный разум «снизу вверх» предпринимались еще с 40х годов специалистами по «нейронной кибернетики». Они стремились разработать самоорганизующуюся систему, способную обучаться интеллектуальному поведению в процессе взаимодействия с окружающим миром, причем компонентами их систем обычно являются модели нервных клеток.

С 1986 года область нейронных сетей вступила в стадию бурного развития. Ежегодно проводятся несколько крупных международных национальных форумов по нейронным сетям, выпускается специальные печатные издания. Возникли фирмы, связанные с нейросетевой технологией, которых к 80ым насчитывалось более 100.

Объем рынка изделий в области нейросетей составил в конце 80х несколько десятков миллионов долларов США. Начался серийный выпуск и эксплуатация основанных на нейросетевой технологии прикладных систем.

Термин «нейрокомпьютер» употребляется в нескольких значениях. Под «нейрокомпьютером» в широком смысле понимают искусственный мозг «разумную систему», которая должна строиться и функционировать по аналогии с мозгом человека.

Слово «нейро» подчеркивает отличие такой системы от традиционного компьютера, который многие также считают «думающей машиной», но который оказался не в состоянии выполнять естественные для поведения живых существ операции восприятия и обработки информации, поступающей из внешнего мира.

Пока еще не созданы действительно «разовые интеллектуальные» системы, способные решать сложные задачи в реальной среде. Поэтому термин «нейрокомпьютер» используют для обозначения всего спектра работ в рамках подхода к построению систем искусственного интеллекта, основанного на моделировании элементов, структур, взаимодействий и функций различных уровней нервной системы.

Для сравнения различных нейрокомпьютеров и оценки их возможностей по эффективной реализации тех или иных нейросетевых моделей используется ряд характеристик.

Максимальный размер нейронной сети, которую можно промоделировать с помощью нейрокомпьютера, определяется количеством нейроподобных элементов и связей, на которые рассчитан нейрокомпьютер, а также возможность изменения их конфигураций.

Быстродействие нейрокомпьютера оценивают количеством связей, которые он способен просмотреть в единицу времени, изменяя их в режиме обучения или учитывая приходящие по связям сигналы в рабочем режиме. Введена аббревиатура CUPS: connection updates per second.

По степени универсальности различают универсальные компьютеры, позволяющие достаточно эффективно реализовать широкий набор нейросетевых моделей, и нейрокомпьютеры, специализированные под узкий класс моделей или под одну модель.

Детальный анализ зарубежных разработок нейрокомпьютеров позволил выделить основные перспективные направления современного развития нейрокомпьютерных технологий: нейропакеты, нейросетевые экспертные системы, СУБД с включением нейросетевых алгоритмов, обработка изображений, управление динамическими системами и обработка сигналов, управление финансовой деятельностью, оптические нейрокомпьютеры, виртуальная реальность.

Сегодня разработками в этой области занимается более 300 зарубежных компаний, причем число их постоянно увеличивается. Среди них такие гиганты как Intel, DEC, ШМ и Motorolla.

Сегодня наблюдается тенденция перехода от программной эмуляции к программноаппаратной реализации нейросетевых алгоритмов с резким увеличением числа разработок СБИС нейрочипов с нейросетевой архитектурой. Резко возросло количество военных разработок, в основном направленных на создание сверхбыстрых, «умных» супервычислителей.

Если говорить о главном перспективном направлении интеллектуализации вычислительных систем, придавая им свойств человеческого мышления и восприятия, то здесь нейрокомпьютеры практически единственный путь развития вычислительной техники.

Многие неудачи на пути совершенствования искусственного интеллекта на протяжении последних 30 лет связаны с тем, что для решения важных и сложных по постановке задач выбирались вычислительные средства, не адекватные по возможностям решаемой задаче, а основном из числа компьютеров, имеющихся под рукой.

При этом, как правило, не решалась задача, а показывалась принципиальная возможность ее решения. Сегодня активное развитие систем MPP создало объективные условия для построения вычислительных систем адекватных по возможностям и архитектуре практически любым задачам искусственного интеллекта.

Архитектура биологической нейронной системы совершенно не похожа на архитектуру машины фон Неймана.

Машина фон Неймана по сравнению с биологической нейронной системой.

Таблица 1.

	Машина фон Неймана	Биологическая нейронная система
Процессор	Сложный	Простой
	Высокоскоростной	Низкоскоростной
	Один или несколь-	Большое количе-
Память	Отделена от процессора	Интегрирована в процессор
	Локализована	Распределенная
	Адресация не по содержанию	Адресация по содержанию
Вычисления	Централизованные	Распределенные
	Последовательные	Параллельные
	Хранимые программы	Самообучение
Надежность	Высокая уязвимость	Живучесть
Специализация	Численные и символьные операции	Проблемы восприятия
Среда функционирования	Строго определенная	Плохо определенная
	Строго ограниченная	Без ограничений

Современные направления развития нейрокомпьютерных технологий.

Реализованные в известных зарубежных нейропакетах нейросетевые парадигмы имеют, по крайней мере, два серьезных недостатка:

- они реализуют нейросетевой алгоритм, не адекватный выбранной задаче;
- достигают локального эффекта на первом этапе использования без возможности улучшения для повышения качества решения задачи.

Определенная общность отечественных методов развития теории нейронных сетей позволила создать единый подход к разработке нейросетевых алгоритмов решения самых разнообразных задач, сформировав новое направление в вычислительной математике нейроматематику.

Эта область связана с разработкой алгоритмов решения математических задач в нейросетевом логическом базисе. Необходимо отметить, что передовая в этом направлении американская школа разработки нейрокомпьютеров уже трижды в истории развития вычислительной техники совершала принципиальные ошибки.

Третья ошибка связана с тем, что в работах американских ученых решение отдельных математических задач в нейросетевом логическом базисе ориентируется на частные нейросетевые парадигмы.

В наших работах общий метод синтеза нейронных сетей позволил создать и развивать в дальнейшем единую методику решения любых математических задач, создавая нейроматематику новый раздел вычислительной математики.

Всегда звучит вопрос, для какого класса задач наиболее эффективно применение того или иного вычислительного устройства, построенного по новым признакам.

По отношению к нейрокомпьютерам ответ на него постоянно меняется в течение уже почти 50 лет.

Долгое время считаюсь, что нейрокомпьютеры эффективны для решения не формализуемых и плохо формализуемых задач, связанных с необходимостью включения в алгоритм решения задач процесса обучения на реальном экспериментальном материале распознавания образов. Конечно, не формализуемые задачи являются важным аргументом использования нейрокомпьютеров.

Однако необходимо помнить, что это всего лишь частная постановка аппроксимации функций, заданных некоторым множеством значений. При этом главное, что для аппроксимации используются не прежние статистические, в частности, регрессионные, а гибкие нелинейные нейросетевые модели.

Обработка изображений. Наиболее перспективными задачами обработки изображений нейрокомпьютерами являются обработка аэрокосмических изображений (сжатие с восстановлением, сегментация, контрастирование и обработка текстур), выделение на изображении движущихся целей, поиск и распознавание на нем объектов заданной формы, обработка потоков изображений, обработка информации в высокопроизводительных сканерах.

Обработка сигналов. В первую очередь это класс задач, связанных с прогнозированием временных зависимостей: прогнозирование финансовых показателей, прогнозирование надежности электродвигателей, упреждение мощности АЭС и прогнозирование надежности систем электропитания на самолетах; обработка траекторных измерений.

При решении этих задач сейчас все переходят от простейших регрессионных и других статистических моделей прогноза к существенно нелинейным адаптивным экстраполирующим фильтрам, реализованным в виде сложных нейронных сетей.

При обработке гидролокационных сигналов нейрокомпьютеры применяются при непосредственной обработке сигнала, распознавании типа

надводной или подводной цели, определении координат цели. Сейсмические сигналы по структуре весьма близки к гидролокационным.

Обработанные нейрокомпьютером позволяют получить с достаточной точностью данные о координатах и мощности землетрясения или ядерного взрыва. Нейрокомпьютеры начали активно использовать при обработке сейсмических сигналов в нефтегазовой разведке. В Международном обществе по нейронным сетям для этого создана специальная группа.

Нейрокомпьютеры в системах управления динамическими объектами.

Это одна из самых перспективных, областей применения нейрокомпьютеров. По крайней мере, США и Финляндия ведут работы по использования нейрокомпьютеров для управления химическими реакторами. В нашей стране им не занимались, в частности, по причине морального устаревания существующих реакторов и нецелесообразности совершенствования их систем управления.

Перспективной считается разработка нейрокомпьютера для управления двигательной установкой гиперзвукового самолета. Фактически единственны вариантом реализации высокопараллельной вычислительной системы управления зеркалами (100400 зеркал) адаптивного составного телескопа сегодня является нейрокомпьютер.

Адаптивные режимы управления этим сложным объектом по критерию обеспечения максимального высокого качества изображения и компенсации атмосферных возмущений может обеспечить мощный нейрокомпьютер, в свою очередь реализующий адаптивный режим собственного функционирования.

Весьма адекватной нейрокомпьютеру является задача обучения нейронной сети выработке точного маневра истребителя. Обучение системы с достаточно слабой нейронной сетью требовало 10 часов на ПК 386.

Тоже можно сказать и о задаче управления роботами: прямая, обратная кинематические и динамические задачи, планирование маршрута движения робота. Переход к нейрокомпьютерам здесь связан в первую очередь с ограниченностью объемов размещения вычислительных систем, а также с необходимостью реализации эффективного управления в реальном масштабе времени.

Можно надеяться, что широкий фронт научных исследований и технических разработок и объединенные усилия ученых разных стран приведут, в конечном счете, к созданию принципиально новых разумных систем.

Нейросетевые экспертные системы. Необходимость реализации экспертных систем в нейросетевом логическом базисе возникает при значительном увеличении числа правил и выводов. Примерами реализации конкретных нейросетевых экспертных систем могут служить система выбора воздушных маневров в ходе воздушного боя и медицинская диагностическая экспертная система для оценки состояния летчика.

Нейрочипы и нейрокомпьютеры. В 1995 году была завершена разработка первого отечественного нейрокомпьютера на стандартной микропроцессорной элементной базе, а сегодня проводится разработка на базе отечественных нейрочипов, в том числе супернейрокомпьютера для решения задач, связанных с системами уравнений математической физики: аэро, гидро, и газодинамики.

Математическая статистика. Нейрокомпьютеры это системы, позволяющие сформировать описания характеристик случайных процессов и совокупности случайных процессов, имеющих в отличие от общепринятого, сложные, зачастую многомодальные или вообще априори неизвестные функции распределения.

Математическая логика и теория автоматов. Нейрокомпьютеры это системы, в которых алгоритм решения задачи представлен логической сетью элементов частного вида нейронов с полным отказом от булевских элементов типа И, ИЛИ, НЕ. Как следствие этого введены специфические связи между элементами, которые являются предметом отдельного рассмотрения.

Теория управления. В качестве объекта управления выбирается частный случай, хорошо формализуемый объект многослойная нейронная сеть, а динамический процесс ее настройки представляет собой процесс решения. При этом практически весь аппарат синтеза адаптивных систем управления переносится на нейронную сеть как частный вид объекта управления.

Вычислительная математика. В отличие от классических методов решения задач нейрокомпьютеры реализуют алгоритмы решения задач, представленные в виде нейронных сетей. Это ограничение позволяет разрабатывать алгоритмы, потенциально более параллельные, чем любая другая их физическая реализация. Множество нейросетевых алгоритмов решения задач составляет новый перспективный раздел вычислительной математики, условно называемый нейроматематикой.

Вычислительная техника. Нейрокомпьютер это вычислительная система с архитектурой MSrMD, в которой реализованы два принципиальных технических решения: упрощен до уровня нейрона

регистра внутренней памяти: происходит сравнение состояния iго элемента регистра Р с состоянием iго элемента строки СМ. Если состояния одинаковы, то изменений не происходит, а если состояния не совпадают, то внутренний элемент принимает значение внешнего.

В блоке 10 происходит уменьшение счетчика цикла *K1* на единицу.

После завершения цикла по К1 определяется номер канала или номер возбужденного мотонейрона Y1.

В блоках 13, 14 к элементам регистра внутренней памяти подключаются соответственно строки 2 и 3 СМ и начинаются такие же циклы *K1* и К2

Подобным образом определяется номер мотонейронов Y2 и Y3.

В блоке 15 заканчивается первый из N тактов возбуждения сети. После окончания третьего цикла К3 в регистре Р накоплено число Уз, которое является статистической смесью всей информации СМ, перенесенной во внутреннюю память под воздействием кода БВГ (U1, U2, U3).

В блоке 16 состояние Y3 переносится в блок динамической регистрации импульсной активности мотонейронов. Он представляет собой счетчик накопитель импульсов N кратного сканирования образа СМ. Каждый накопленный импульс отображается на экране дисплея коды знаком (*). Над столбиком звездочек изображается номер мотонейрона и десятичное число, показывающая степень возбуждения.

Если Уз1 <0, то переменная SY принимает значение 1 и на экране, на выходе "О" не читается знак "*" и число "1".

Если Уз1=0, аналитический процесс происходит на выходе "1".

Если Уз1>0, то после сравнения Уз с числом 2, выбирается в выход либо "1" либо "2",либо "3" и туда заносится информация в виде знака "*" и знака.

В блоке 17 проверяется условие N=0.

В блоке 18_печатаются исходные данные: образ сенсорной матрицы [S1,...,S9], код БВГ [U1, U2, U3] и значения NS.

В данной программе сеть нейронов построена так, что на выходе подключено четыре мотонейрона: накопленное количество импульсов на выходе.

Контрольные вопросы

1. Назовите основные блоки нейрокомпьютера "ЭМБРИОН".
2. Какую функцию выполняет сенсорная матрица (СМ)?
3. Какую функцию выполнят блок выдвижения гипотез (БВГ)?
4. Что мы называем энергетическим потенциалом сети?

5. По какой формуле рассчитывается вектор Z и как он называется?
6. Из каких циклов состоит алгоритм работы бытового нейрокомпьютера (БНК)?
7. Как происходит подсчет количества импульсов и распределение частот мотонейронов при заданном параметре NS?
8. Какую структуру памяти имеет нейрокомпьютер "ЭМБРИОН".
9. Какие программные реализации выпускаемых нейрокомпьютеров вам известны?
10. Какие основные перспективные направления современного развития нейрокомпьютерных технологий вы знаете?

Содержание отчёта

1. Использовать листы формата А4
2. Использовать шрифта Times New Roman, размер кегля 14
3. Интервал 1,5
4. Написать тему, цель и задачи практического работы
5. Выполнить задание
6. Написать выводы и ответить письменно на вопросы.

Практическое занятие № 9

Тема: Изучение формирования и работы пакета OpenMP

Цели работы: Изучение формирования и работы пакета OpenMP на языке C++

Теоретическая часть

OpenMP — это библиотека для параллельного программирования вычислительных систем с общей памятью (дальше кратко описано что это за системы). Официально поддерживается Си, C++ и Фортран, однако можно найти реализации для некоторых других языков, например Паскаль и Java.

Библиотека активно развивается, в настоящий момент актуальный стандарт версии 4.5 (выпущен в 2015 году), однако уже идет разработка версии 5.0. В тоже время, компилятор Microsoft C++ поддерживает только версию 2.0, а gcc — версию 4.0.

Библиотека OpenMP часто используется в математических вычислениях, т.к. позволяет очень быстро и без особого труда распараллелить вашу программу. При этом, идеология OpenMP не очень хорошо подойдет, скажем, при разработке серверного ПО (для этого существуют более подходящие инструменты).

Вычислительные системы. Идеология OpenMP

Существует множество разновидностей параллельных вычислительных систем — многоядерные/многопроцессорные компьютеры, кластеры, системы на видеокартах, программируемые интегральные схемы и т.д.

Библиотека OpenMP подходит только для программирования систем с общей памятью, при этом используется параллелизм потоков. Потоки создаются в рамках единственного процесса и имеют свою собственную память. Кроме того, все потоки имеют доступ к памяти процесса. Схематично это показано на рисунке 20:

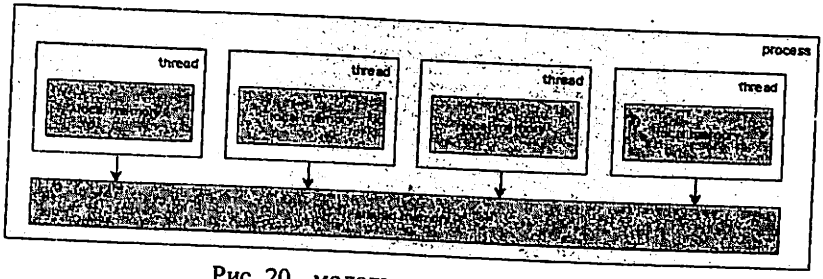


Рис. 20 - модель памяти в OpenMP

Для использования библиотеки OpenMP вам необходимо подключить заголовочный файл "omp.h", а также добавить опцию сборки –fopenmp (для компилятора gcc) или установить соответствующий флажок в настройках проекта (для Visual Studio).

После запуска программы создается единственный процесс, который начинается выполняться, как и обычная последовательная программа. Встретив параллельную область (задаваемую директивой #pragma omp parallel) процесс порождает ряд потоков (их число можно задать явно, однако по умолчанию будет создано столько потоков, сколько в вашей системе вычислительных ядер).

Границы параллельной области выделяются фигурными скобками, в конце области потоки уничтожаются. Схематично этот процесс изображен на рисунке 23:

```
#include "omp.h"

int main() {
  // A - single thread
#pragma omp parallel
  {
    // B - many threads
  }
  // C - single thread
#pragma omp parallel
  {
    // D - many threads
  }
  // E - single thread
}
```

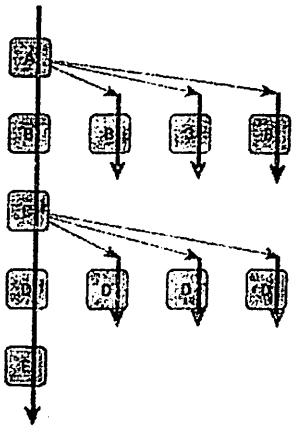


Рис.23 - директива omp parallel

Черными линиями на рисунке показано время жизни потоков, а красными — их порождение. Видно, что все потоки создаются одним (главным) потоком, который существует все время работы процесса. Такой поток в OpenMP называется master, все остальные потоки многократно создаются и уничтожаются. Стоит отметить, что директивы parallel могут быть вложенными, при этом в зависимости от настроек (изменяются функцией omp_set_nested) могут создаваться вложенные потоки.

OpenMP может использоваться на кластерной архитектуре, но не самостоятельно, т.к. загрузить весь кластер она не может (для этого нужно

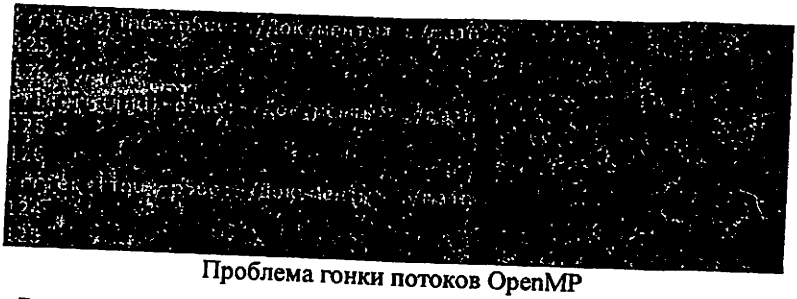
создавать процессы, использовать инструменты типа MPI). Однако, если узел кластера многоядерный — то использование OpenMP может существенно поднять эффективность. Такое приложение будет являться «гибридным».
Синхронизация — критические секции, atomic, barrier

Все переменные, созданные до директивы parallel являются общими для всех потоков. Переменные, созданные внутри потока являются локальными (приватными) и доступны только текущему потоку.

При изменении общей переменной одновременно несколькими потоками возникает состояние гонок (мы не можем гарантировать какой-либо конкретный порядок записи и, следовательно, результат) — это проблема и допускать такое нельзя. Такая же проблема возникает когда один поток пытается читать переменную в то время, как другой ее изменяет. Ситуацию поясняет следующий пример:

```
#include "omp.h"
#include <iostream>
int main() {
 int value = 123; #pragma omp parallel
 {value++;#pragma omp critical
  {std::cout << value++ << std::endl;
  }
 }
}
```

Программа описывает переменную value, общую для всех потоков. Каждый поток увеличивает значение переменной, а затем выводит полученное значение на экран. запустив ее на двухъядерном компьютере, получили следующие результаты:



Проблема гонки потоков OpenMP

Видно, что чаще всего сначала каждый из потоков увеличит значение переменной, а затем они по-очереди выведут результаты (при этом каждый из них еще раз увеличивает значение), но в некоторых случаях порядок выполнения будет другим.

В этом примере сейчас интересует, при попытке одновременного увеличения значения value программа может вести себя как угодно — увеличить только один раз или вовсе завершиться аварийно.

Для решения проблемы существует директива critical, пример ее использования также показан выше. Разделяемым ресурсом в этом примере является не только память (размещенные в ней переменные), но и консоль (в которую потоки выводят результаты).

В примере гонки возникают при инкременте переменной, но не при выводе на экран, т.к. операции с cout помещены в критическую секцию. В критической секции в один момент времени может находиться только один поток, остальные ожидают ее освобождения.

Правилом хорошего тона считается, если критическая секция содержит обращения только к одному разделяемому ресурсу (в примере секция не только выводит данные на экран, но и выполняет инкремент — это не очень хорошо, в общем случае).

Для ряда операций более эффективно использовать директиву atomic, чем критическую секцию. Она ведет себя также, но работает чуть быстрее. Применять ее можно для операций префиксного/постфиксного инкремента/декремента и операции типа X BINOP = EXPR, где BINOP представляет собой не перегруженный оператор +, *, -, /, &, ^, |, <<, >>. Пример использования такой директивы:

```
#include "omp.h"
#include <iostream>
int main() {
 int value = 123; #pragma omp parallel
 { #pragma omp atomic
  value++;#pragma omp critical (cout)
  {std::cout << value << std::endl;}
 }
}
```

Тут же показана возможность именования критических секций — очень рекомендую ей пользоваться всегда. Дело в том, что все безымянные секции рассматриваются как одна (очень большая) и если вы не дадите им явно имена, то только в одной из этих секций в один момент времени будет один поток. Остальные будут ждать.

Имя секции должно подсказывать программисту к какому виду ресурса она относится — в приведенном примере таким ресурсом является поток вывода на экран (cout).

Несмотря на то, что каждая операция над общими данными в последнем примере размещена в критической секции или является атомарной, в нем есть проблема, т.к. порядок выполнения этих операций по

прежнему не определен. Запустив программу раз 20 мне удалось получить на экране не только "125 125", но и "124 125".

Если мы хотим чтобы сначала каждый поток увеличил значение, а затем они вывели их на экран — можем использовать директиву barrier:

```
#pragma omp parallel
{#pragma omp atomic
   value++;
 #pragma omp barrier
 #pragma omp critical (cout)
 { std::cout << value << std::endl;}
}
```

Поток, завершив свою часть вычислений доходит до директивы barrier и ждет пока все потоки дойдут до этой же точки. Дождавшись последнего, потоки продолжают выполнение.

Теперь в программе нет никаких проблем с синхронизацией, но обратите внимание, что все потоки выполняют одну и туже работу (описанную внутри параллельной области), пусть и параллельно. При таком раскладе программа не начнет работать быстрее, нам необходимо распределить между потоками решаемые задачи, сделать это можно различными способами...

Разделение задач между потоками

Параллельный цикл

Самый популярный способ распределения задач в OpenMP — параллельный цикл. Не секрет, что программы почти всю свою жизнь, проводят выполняя циклы, при этом если между итерациями цикла нет зависимостей — то цикл называется векторизуемым (его итерации можно поделить между потоками и выполнить независимо друг от друга).

Параллельный цикл позволяет задать опцию schedule, изменяющую алгоритм распределения итераций между потоками. Всего поддерживается 3 таких алгоритма. Далее полагаем, что у нас p потоков выполняют n итераций:

Опции планирования:

schedule(static) — статическое планирование. При использовании такой опции итерации цикла будут поровну (приблизительно) поделены между потоками. Нулевой поток получит первые np итераций, первый — вторые и т.д.;

schedule(static, 10) — блочно-циклическое распределение итераций. Каждый поток получает заданное число итераций в начале цикла, затем (если остались итерации) процедура распределения продолжается. Планирование выполняется один раз, при этом каждый поток «узнает» итерации которые должен выполнить;

schedule(dinamic), schedule(dynamic, 10) — динамическое планирование. По умолчанию параметр опции равен 1. Каждый поток получает заданное число итераций, выполняет их и запрашивает новую порцию. В отличии от статического планирования, выполняется многократно (во время выполнения программы). Конкретное распределение итераций между потоками зависит от темпов работы потоков и трудоемкости итераций;

schedule(guided), schedule(guided, 10) — разновидность динамического планирования с изменяемым при каждом последующем распределении числе итераций.

Распределение начинается с некоторого начального размера, зависящего от реализации библиотеки до значения, задаваемого в опции (по умолчанию 1). Размер выделяемой порции зависит от количества еще нераспределенных итераций

В большинстве случаев самым оптимальным вариантом является static, т.к. выполняет распределение единственный раз, играть с этим параметром имеет смысл если в вашей задаче сильно отличается трудоемкость итераций. Например, если вы считаете сумму элементов квадратной матрицы, расположенных ниже главной диагонали — то static даст не лучший результат, т.к. первый поток выполнит значительно меньше операций и будет простаивать.Итак, параллельно вычислим сумму элементов массива

```
int sum_arr(int *a, const int n) {  int sum = 0;
#pragma omp parallel reduction (+: sum)
 {#pragma omp for
  for (int i = 0; i < n; ++i)
   sum += a[i];}
 return sum;
}
```

Параллельные задачи (parallel tasks)

Параллельные задачи — это более гибкий механизм, чем параллельный цикл. Параллельный цикл описывается внутри параллельной области, при этом могут возникнуть проблемы.

Например, мы написали параллельную функцию вычисления суммы элементов одномерного массива, и нашу функцию решили применить для вычисления суммы элементов матрицы, но сделать это также параллельно. Получится вложенный параллелизм. Если (теоретически) наш код запущен на 8 ядрах — то фактически будет создано 64 потока. Ну а если кому-нибудь придет в голову идея делать параллельно еще что-нибудь?

Проблема параллельного цикла в том, что число создаваемых потоков зависит от того какие функции распараллелены и как они друга друга вызывают. Очень сложно все это отслеживать и, тем более, поддерживать.

Решение проблемы — параллельные задачи, которые не создают поток, а лишь выполняют добавление задачи в очередь, освободившийся поток выбирает задачу из пула.

Параллельные секции

Механизм параллельных секций видится мне достаточно низкоуровневым. Тем не менее, он полезен в ряде случаев. Как было отмечено выше, параллельный цикл можно применять только в случаях, если итерации цикла не зависят друг от друга, т.е. тут нельзя:

```
for (int i = 1; i < n; ++i)
  a[i] = a[i-1]+1;
```

Если же у нас в программе появляется несколько фрагментов, не зависящих друг от друга, но имеющий зависимости внутри себя — то их распараллеливают с помощью механизма параллельных секций:

```
for (int i = 1; i < n; ++i)
  a[i] = a[i-1]+1;
{
#pragma omp sections
  {
#pragma omp section
    {
          for (int i = 1; i < n; ++i)
       a[i] = a[i-1]+1;}
#pragma omp section
    {
          for (int i = 1; i < n; ++i)
       b[i] = b[i-1]+1;}
  }
}
```

Очень важно — не пытайтесь распараллелить рекурсивные функции с помощью секций (используйте для этого задачи).

OpenMP поддерживается многими современными компиляторами.

Компиляторы Sun Studio поддерживают спецификацию OpenMP 2.5 с поддержкой операционной системы Solaris; поддержка Linux запланирована на следующий выпуск.

Эти компиляторы создают отдельную процедуру из исходного кода, располагающегося под директивой parallel, а вместо самой директивы вставляют вызов процедуры __mt_MasterFunction_ библиотеки libmtsk, передавая ей адрес искусственно созданной.

Таким образом, разделяемые (shared) данные могут быть переданы последней по ссылке, а собственные (private) объявляются внутри этой процедуры, оказываясь независимыми от своих копий в других потоках.

Процедура __mt_MasterFunction_ создает группу потоков (количеством 9 в приведенном выше примере на языке C), которые будут выполнять код конструкции parallel, а вызвавший её поток становится главным в группе. Затем главный поток организовывает работу подчиненных потоков, после чего начинает выполнять пользовательский код сам. Когда код будет выполнен, главный поток вызывает процедуру _mt_EndOfTask_Barrier_, синхронизирующую его с остальными.

Visual C++ 2005 и 2008 поддерживает OpenMP 2.0 в редакциях Professional и Team System, 2010 — в редакциях Professional, Premium и Ultimate, 2012 — во всех редакциях.

В GCC начиная с версии 4.2 реализована поддержка OpenMP для Си, C++ и Фортрана (на базе gfortran), а некоторые дистрибутивы (такие как Fedora Core 5) включили поддержку в GCC 4.1. В Clang и LLVM 3.7 поддерживается OpenMP 3.1..

Intel C++ Compiler, Intel Fortran Compiler и Intel Parallel Studio поддерживают версию OpenMP 3.0, а также Intel Cluster OpenMP для программирования в системах с распределённой памятью. Существуют также реализации в компиляторах IBM XL compiler, PGI (Portland group), Pathscale, HP.

Контрольные вопросы:

1. Что такое OpenMp?
2. Для чего применяется библиотека OpenMP
3. Какие языки программирования поддерживают OpenMP?
4. Перечислите плюсы и минусы использования Параллельного цикла?

5. Какие различия между параллельным циклом и секцией?

Содержание отчёта

1. Использовать листы формата А4
2. Использовать шрифта Times New Roman, размер кегля 14
3. Интервал 1,5
4. Написать тему, цель и задачи практического работы
5. Выполнить задание
6. Написать выводы и ответить письменно на вопросы.

Практическое занятие №10

Изучение работы пакета MPI

Цель работы: изучить структуру стандарта программирования системы MPI (Message Passing Interface) и продемонстрировать ряд функций её библиотеки.

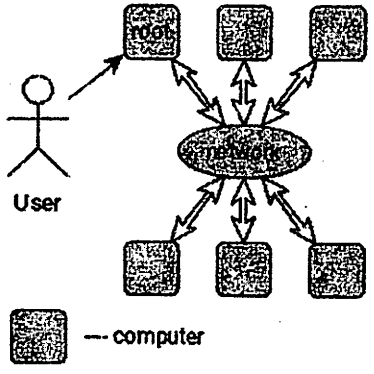
Теоретическая часть

Параллельное программирование — очень актуальное направление, т.к. большинство современных вычислительных устройств (включая телефоны) являются многоядерными или многопроцессорными. В предыдущей записи я публиковал учебник по OpenMP, однако OpenMP позволяет программировать только системы с общей памятью — в большей части, многоядерные компьютеры.

Основной проблемой таких систем является плохая масштабируемость — не так легко и дешево увеличить число ядер в компьютере.

Другой класс параллельных ЭВМ — системы с распределенной памятью, к которым, в частности, относятся кластеры. Они представляют собой набор соединенных сетью компьютеров.

Такая система гораздо лучше мастабируется — не составит особого труда купить и подключить в сеть дополнительную сотню компьютеров. Число вычислительных узлов в кластерах измеряется тысячами.



Взаимодействуют узлы кластера с помощью передачи сообщений по сети, поэтому стандартом программирования таких систем является MPI (Message Passing Interface). Библиотека MPI предоставляет программисту

огромный выбор способов передать сообщение между узлами, в этой статье я постараюсь описать основные способы на простых примерах.

Ключевые особенности MPI

Допустим, есть у нас кластер. Чтобы программа начала на нем выполняться, ее необходимо **скопировать** на каждый узел, **запустить** и **установить связь** между процессами. Эту работу берет на себя утилита *mpirun* (под Linux) или *mpiexec* (под Windows), так например, чтобы запустить 5 процессов достаточно написать:
mpirun -np 5 path/your_mpi_program

Однако программа должна быть написана определенным образом. Вообще, технология MPI позволяет как использовать модель **SPMD** (Single Process, Multiple Data), так и **MPMD** [1], в этой статье я рассматривают только первый вариант.

Далее по тексту узел и процесс будут означать одно и тоже, хотя на одном узле может быть создано несколько процессов (именно так я делаю при запуске примеров статьи, т.к. отлаживаю их на персональном компьютере). Это вводная статья, поэтому тут не пойдет речь о *коммуникаторах*, в которые могут группироваться процессы.

Суть SPMD заключается в том, что для решения задачи запускается множество **одинаковых** процессов. На приведенном выше рисунке видно, что пользователь (который вводит данные и хочет получить результат) взаимодействует только с одним узлом кластера.

Такой узел называется **root** и логика его работы должна отличаться, ведь он должен не только взаимодействовать с пользователем, но и, получив исходные данные, выполнить их рассылку остальным процессам. Каждый процесс имеет свой номер (в MPI принят термин «**ранг**«) в рамках каждого коммуникатора, при этом *у root ранг обычно равен нулю*.

Процессы **обмениваются сообщениями**, каждое из которых помимо *номеров процесса отправителя и получателя* имеет *тег*, а также *тип* и *количество передаваемых элементов*.

Тег представляет собой целое число, с помощью которого программа может отличать одни сообщения от других. В силу того, что сегодня мы рассмотрим очень простые примеры, тег нам не особо пригодится (можно было бы везде использовать, скажем, «ноль» вместо тега). Все поступающие процессу сообщения помещаются в **очередь**, из которой могут быть извлечены в соответствии с запрашиваемыми параметрами (тегом, отправителем и т.п.).

В связи с тем, что MPI-программа обладает множеством особенностей, компилироваться она должна специальным компилятором. Под Linux для этого используется mpic++, а под Windows можно применять расширение для Microsoft Visual Studio. Для сборки примеров статьи под Linux я использовал примерно следующую команду:

mpic++ main.cpp -o main

Разобравшись с особенностями технологии, перейдем к примерам. В этой статье мы будем пытаться посчитать сумму элементов массива — несмотря на простоту, эта задача позволяет продемонстрировать самые различные способы передачи данных между узлами.

Операции точка-точка MPI

Блокирующие операции — MPI_Send, MPI_Recv, MPI_Probe

Операции точка-точка позволяют передавать данные между парой узлов. Самые простые функции этого вида — MPI_Send и MPI_Recv, выполняющие передачу и прием сообщения, соответственно:

MPI_Send(**void*** buf, **int** count, MPI_Datatype datatype, **int** dest, **int** tag, MPI_Comm comm)

MPI_Recv(**void*** message, **int** count, MPI_Datatype datatype, **int** source, **int** tag, MPI_Comm comm, MPI_Status* status)

Функция MPI_Send выполняет передачу count элементов типа datatype, начиная с адреса определенного buf, процессу с номером dest в коммуникаторе comm. При этом сообщению присваивается некоторый тег.

Функция MPI_Recv выполняет прием, при этом она ожидает появления во входном буфере сообщения с заданными параметрами (выполнение программы не продолжится до того, как такое сообщение поступит). Функция также возвращает статус, который может содержать код ошибки.

Рассмотрим первый пример:

```
#include "mpi.h"

#include <iostream>

using namespace std;

const int Tag = 0;

const int root = 0;

double sum_array(double *array, int n) {

double sum = 0;
```

```
for (int i = 0; i < n; ++i) {
sum += array[i];
}
return sum;
}
int main() {
int rank, commSize;
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &commSize);
double *arr, sum = 0, buffer;
int n;
MPI_Status status;
if (root == rank) {
cout << "n : ";
cin >> n;
arr = new double[n];
for (int i = 0; i < n; ++i)
cin >> arr[i];
int partSize = n/commSize;
int shift = n%commSize;
for (int i = root+1; i < commSize; ++i) {
MPI_Send(arr + shift + partSize*i, partSize, MPI_DOUBLE, i, Tag,
MPI_COMM_WORLD);
}
sum = sum_array(arr, shift + partSize);
for (int i = root+1; i < commSize; ++i) {
MPI_Recv(&buffer, 1, MPI_DOUBLE, i, Tag, MPI_COMM_WORLD, &status);
```

```
sum += buffer;
}
}
else {
MPI_Probe(root, Tag, MPI_COMM_WORLD, &status);
MPI_Get_count(&status, MPI_DOUBLE, &n);
arr = new double[n];
MPI_Recv(arr, n, MPI_DOUBLE, root, Tag, MPI_COMM_WORLD, &status);
sum = sum_array(arr, n);
MPI_Send(&sum, 1, MPI_DOUBLE, root, Tag, MPI_COMM_WORLD);
}
delete[] arr;
cout << rank << " : " << sum << endl;
MPI_Finalize();
}
```

Все обращения к функциям MPI должны размещаться между MPI_Init и MPI_Finalize. В качестве коммуникатора в этом примере используется MPI_COMM_WORLD, который включает в себя все процессы, запущенные для текущей задачи с помощью mpirun.

Каждый процесс получает свой ранг с помощью вызова MPI_Comm_rank, количество процессов в коммуникаторе возвращает функция MPI_Comm_size.

В теле программы явно выделяется два блока кода — один выполняется главным процессом, а другой — остальными. Главный процесс занимается вводом/выводом данных, и их распределением между остальными узлами. Все остальные процессы ожидают от главного часть массива, вычисляют сумму элементов и передают результат назад.

Чтобы принять массив, дочерние процессы должны сначала выделить память, но для этого необходимо знать сколько именно элементов им будет передано — сделать это можно с помощью функции MPI_Probe, которая работает также как MPI_Send (блокирует процесс до поступления в очередь сообщения с заданными параметрами), но не удаляет сообщение из очереди,

а лишь возвращает его статус. Структура типа MPI_Status содержит тип элементов и их количество.

Операции MPI_Send, MPI_Probe и MPI_Recv являются синхронными (блокирующими), т.к. останавливают выполнение основного потока процесса до тех пор, пока не выполнится какое-либо действие (данные не будут записаны в сокет или не поступит сообщение с требуемыми параметрами).

Асинхронные операции — MPI_Isend

Приведенный выше пример работает прекрасно, но бутылочным горлышком является нулевой процесс, т.к. перед тем как приступить к обработке своей части массива root должен передать данные всем остальным процессами.

Немного улучшить картину можно с помощью неблокирующих (асинхронных) операций — одной из них является MPI_Isend.

Функции неблокирующего обмена имеют такие же аргументы как и их блокирующие аналоги, но в качестве дополнительного параметра принимают параметр типа MPI_Request. Чтобы нулевой процесс в приведенном выше примере работал асинхронно достаточно изменить лишь этот фрагмент кода:

```
for (int i = root+1; i < commSize; ++i) {
MPI_Isend(arr + shift + partSize*i, partSize,
MPI_DOUBLE, i, Array, MPI_COMM_WORLD, &request);
MPI_Request_free(&request);
}
```

Теперь функция передачи возвращает управление немедленно, сама же передача происходит параллельно с выполнением других команд процесса. Узнать о ходе выполнения асинхронной операции мы сможем с помощью специальных функций:

```
int MPI_Wait(MPI_Request *request, MPI_Status *status);
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
```

Функция MPI_Wait является блокирующей — она останавливает выполнение процесса до тех пор, пока передача, связанная с request не будет завершена. Функция MPI_Test является неблокирующей (не дожидается окончания выполнения операции) — о том была завершена операция или нет, она сигнализирует с помощью флага (true — завершена).

В нашем примере в использовании функций MPI_Wait и MPI_Test нет необходимости, т.к. синхронно выполняется сбор результатов вычислений процессов. Главный процесс инициирует передачу данных и приступает к

обработке своей части массива, после этого синхронно ожидает поступления данных от каждого по очереди процесса. Кстати, асинхронной может быть не только передача, но и прием данных — функция MPI_Irecv.

Чтобы избежать ошибок, необходимо представлять как именно может быть реализована работа неблокирующих функций. Например, MPI_Isend инициирует передачу данных, которая выполняется в отдельном потоке параллельно. По окончанию передачи этот поток должен изменить переменную request. Это значит, что такой код вполне может привести к ошибкам:

```
for (int i = root+1; i < commSize; ++i) {
MPI_Request request;
MPI_Isend(arr + shift + partSize*i, partSize, MPI_DOUBLE,
i, Array, MPI_COMM_WORLD, &request);
}
```

Тут на каждой итерации цикла создается экземпляр переменной типа MPI_Request, инициируется передача и объект разрушается (память освобождается).

Это значит, что поток, выполняющий передачу обратится к памяти, которая будет освобождена (и возможно уже повторно распределена для других целей), а значит — программа будет вести себя непредсказуемо. Вывод — объект request должен существовать до завершения асинхронной передачи.

Вызов функции MPI_Request_free необходим в случаях если не требуется ждать окончания асинхронной передачи и при этом хочется использовать один экземпляр MPI_Request — в нашем случае один и тот же объект используется для передачи данных всем дочерним процессам. За более детальной информацией о работе асинхронных операций MPI предлагаю обратиться к стандарту (раздел 3.7 Nonblocking Communication).

Буферизованная передача — MPI_Bsend

В стандарте MPI сказано, что функция MPI_Send вернет управление после того, как сообщение будет записано в *выходной буфер*, я же выше писал что это *сокет* (пусть это не совсем точно) — стандарт в этом месте дает некоторую свободу, возможны различные реализации.

Однако, «выходной буфер» в стандарте — это однозначно не область в оперативной памяти, т.к. для работы с буфером в ОЗУ предусмотрена другая функция — MPI_Bsend.

Функция MPI_Send записывает данные в сокет и только после этого возвращает управление. Функция MPI_Isend вернет управление сразу, но нам все равно нельзя будет изменять передаваемые данные, пока они не будут записаны в сокет.

При этом, работа с сокетом выполняется медленнее, чем с оперативной памятью — поэтому ускорить работу программы в ряде случаев можно путем копирования данных в некоторый буфер в оперативной памяти и передачей данных из этого буфера. Изменять отправляемые данные мы сможем сразу после того, как они будут скопированы. Примерно так и работает функция MPI_Bsend.

Сложность реализации такой операции вручную заключается также в том, что после отправки данных память из под буфера нужно освободить.

Чтобы выделенная область памяти использовалась в качестве буфера при передачи в буферизованном режиме — необходимо использовать функцию MPI_Buffer_attach.

Буфер может состоять из нескольких «присоединенных» областей. В разделе 3.6.1 стандарта говорится о том, что буфер может быть устроен как циклическая очередь сообщений, отправленные сообщения из буфера удаляются, позже на их место могут быть записаны другие сообщения.

При использовании буферизованного режима исходный код будет не сильно отличаться от приведенного выше, ниже приведен только фрагмент, которого коснулись изменения:

```
int message_buffer_size = n*sizeof(double) + MPI_BSEND_OVERHEAD;
double* message_buffer = (double*)malloc(message_buffer_size);
MPI_Buffer_attach(message_buffer, message_buffer_size);
int shift = n%commSize;
for (int i = root+1; i < commSize; ++i) {
MPI_Bsend(arr + shift + partSize*i, partSize, MPI_DOUBLE, i, Tag,
MPI_COMM_WORLD);
}
sum = sum_array(arr, shift + partSize);
for (int i = root+1; i < commSize; ++i) {
```

```
MPI_Recv(&buffer, 1, MPI_DOUBLE, i, Tag, MPI_COMM_WORLD, &status);
sum += buffer;
}
MPI_Buffer_detach(message_buffer, &message_buffer_size);
free(message_buffer);
```

Перед использованием функции выделяется память под буфер, т.к. в этой памяти размещается очередь — следовательно есть некоторые издержки (размер которых зависит от конкретной реализации стандарта), поэтому необходимо выделять чуть больше памяти (MPI_BSEND_OVERHEAD).

Выделенная область прикрепляется к буферу, затем используется MPI_Bsend (аргументы функции полностью совпадают с MPI_Send, буфер в качестве аргумента не передается, т.к. он является глобальным для процесса). После того, как все сообщения будут переданы — буфер можно отсоединить, а память освободить.

Коллективные операции. Пример использования MPI_Reduce

Коллективные операции выполняются всеми процессами указанного коммуникатора. Ниже приведена картинка из стандарта, на которой показана суть некоторых операций:

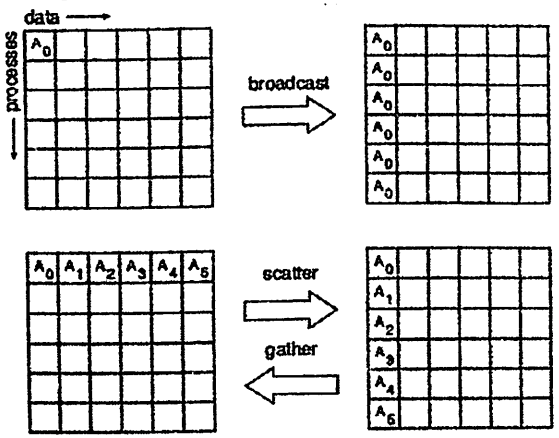


Рис. 22 – Коллективная операция

Верхняя часть схемы иллюстрирует операцию **MPI_Bcast**, которая позволяет передать некоторые данные с одного узла кластера на все остальные. Нижняя — соответствует операциям MPI_Scatter и MPI_Gather.

Если у нас на узле U есть массив из N элементов и его части необходимо передать на P узлов кластера — можно использовать функцию **MPI_Scatter**. Проблем не возникнет если N делится нацело на P, т.к. при выполнении MPI_Scatter все узлы получат одинаковое количество элементов. Обратную операцию выполняет **MPI_Gather**, т.е. собирает данные со всех P узлов на узел U.

Эти операции являются синхронными и используют MPI_Send (это закреплено стандартом), однако существуют асинхронные аналоги — MPI_Ibcast, MPI_Igather и MPI_Iscatter.

Операция MPI_Bcast теоретически (зависит от реализации библиотеки) может работать более эффективно и выполняться за O(log(n)) операций вместо O(n).

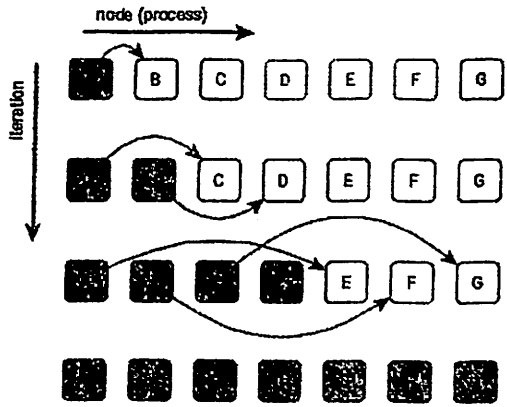


Рис.23 - Операция MPI_Bcast

На приведенной схеме цветом выделен узел, на котором находятся передаваемые данные. В начале работы такой узел один. После первой передачи данные есть уже на двух узлах, оба они могут участвовать в передачи. При реализации такой схемы для передачи данных на 1000 узлов будет достаточно 10 операций.

Операция MPI_Reduce не просто передает данные, но и выполняет над ними заданную операцию. В нашем примере применить ее можно вместо сбора результатов вычисления сумм:

```
if (root == rank) {
cout << "n : ";
cin >> n;
arr = new double[n];
for (int i = 0; i < n; ++i)
cin >> arr[i];
int partSize = n/commSize;
int shift = n%commSize;
for (int i = root+1; i < commSize; ++i) {
MPI_Send(arr + shift + partSize*i, partSize, MPI_DOUBLE, i, Tag,
MPI_COMM_WORLD);
}
sum = sum_array(arr, shift + partSize);
}
else {
MPI_Probe(root, Tag, MPI_COMM_WORLD, &status);
MPI_Get_count(&status, MPI_DOUBLE, &n);
arr = new double[n];
MPI_Recv(arr, n, MPI_DOUBLE, root, Tag, MPI_COMM_WORLD, &status);
sum = sum_array(arr, n);
}

double global_sum = 0;

MPI_Reduce(&sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, root,
MPI_COMM_WORLD);

if (rank == root) {

cout << "sum: " << global_sum << endl;

}
```

Операция
MPI_Reduce
может выполняться не только над числами, но и над массивами (при этом будет применена к каждому его элементу отдельно).

Контрольные вопросы:

1. Ключевые особенности MPI
2. Операции точка-точка MPI
3. Асинхронные операции — MPI_Isend
4. Буферизованная передача — MPI_Bsend
5. Коллективные операции. Пример использования MPI_Reduce

Содержание отчёта

1. Использовать листы формата А4
2. Использовать шрифта Times New Roman, размер кегля 14
3. Интервал 1,5
4. Написать тему, цель и задачи практического работы
5. Выполнить задание
6. Написать выводы и ответить письменно на вопросы.

ЗАКЛЮЧЕНИЕ

Архитектура компьютера – это логическая архитектура и структура аппаратных и программных ресурсов вычислительной системы. Архитектура заключает в себе требования к функциональности и принципы организации основных узлов компьютера.

Внешняя архитектура современного персонального компьютера представляет собой соединение монитора, клавиатуры, мыши и акустической системы к системному блоку.

Внутренняя архитектура современного персонального компьютера определяется схемой его чипсета, набором микросхем, спроектированных для совместной работы с целью выполнения набора каких-либо функций. компьютерах Чипсет в компьютере выполняет роль связующего компонента, обеспечивающего совместное функционирование подсистем памяти, ЦПУ, ввода-вывода и других. Выбор типа чипсета зависит от процессора, с которым он работает, и определяет разновидности внешних устройств (видеокарты, винчестера и др.).

Работа в данном направлении, затрагивая, в первую очередь, программное обеспечение, потребует и создания компьютера определенной архитектуры, используемых в системах управления базами знаний, - компьютеров баз знаний, а так же других подклассов компьютера. При этом компьютер должна обладать способностью к обучению, производить ассоциативную обработку информации и вести интеллектуальный диалог при решении конкретных задач.

В заключение отметим, что ряд названных вопросов реализован в перспективных компьютеров пятого поколения либо находится в стадии технической проработки, другие - в стадии теоретических исследований и поисков.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

Основная литература:

1. Таненбаум Э., Остин Г. Архитектура компьютера. 6-е изд. СПб.: Питер, 2014.
2. Абросимов, Л. И. Базисные методы проектирования и анализа сетей ЭВМ. Учебное пособие / Л.И. Абросимов. - М.: Университетская книга, 2015. - 248 с
3. Авдеев, В. А. Периферийные устройства. Интерфейсы, схемотехника, программирование. Учебное пособие / В.А. Авдеев. - М.: ДМК Пресс, 2016. - 848 с.
4. Гуров В.В. - Архитектура микропроцессоров - Национальный Открытый Университет "ИНТУИТ" - 2016 - 327с. - ISBN: 978-5-9963-0267-3
5. Гергель В.П., Мееров И.Б., Бастраков С.И. - Введение в принципы функционирования и применения современных мультиядерных архитектур "ИНТУИТ" - 2016 - 407с

Дополнительная литература:

1. Новожилов О. П. - АРХИТЕКТУРА ЭВМ И СИСТЕМ. Учебное пособие для академического бакалавриата - М.:Издательство Юрайт - 2018 - 527с. - ISBN: 978-5-534-02626-9
2. Максимов, Н.В. Архитектура ЭВМ и вычислительных систем : учебник / Н.В.

Веб-сайты:

1. http://www.infl.info. Планета информатики.
2. http://perscom.ru. ПерсКом.
3. http://all-ht.ni. Информационный сайт о высоких технологиях.
4. Top 500 Supercomputer Sites (http://www.top500.org).
5. Суперкомпьютеры Top 50 (http://supercomputers.ru).

ОГЛАВЛЕНИЕ

Формат 60x84 1/16. Печ.лист 6.
Заказ №154. Тираж 40.
Отпечатано в «Редакционно издательском»
отделе при ТУИТ.
Ташкент ул. Амир Темур, 108.