1. Информация. Меры Хартли, Шеннона.

Различные формулировки понятия “информация”:

- В законодательном плане: сведения о лицах, предметах, фактах, событиях, явлениях и процессах независимо от формы их представления (ФЕДЕРАЛЬНЫЙ ЗАКОН Об информации, информатизации и защите информации);
- В концептуальном плане: универсальная субстанция, пронизывающая все сферы человеческой деятельности, служащая проводником знаний и мнений, инструментом общения, взаимопонимания и сотрудничества, утверждения стереотипов мышления и поведения (ЮНЕСКО); бесконечный законопроцесс триединства энергии, движения и массы в пространстве и во времени с различными плотностями кодовых структур бесконечно-беспредельной Вселенной. Информация внутри нас, информация вне нас. Есть законы существования этой информации внутри и вне нас (Международная Академия Информатизации)
- В математике, системном анализе — любая сущность, которая вызывает изменения в некоторой информационно-логической (инфологической — состоящей из сообщений, данных, знаний. абстракций и т. д.) модели, представляющей систему
- В термодинамике:
  1. отрицание энтропии, отражение меры хаоса в системе (Бриллюэн);
  2. передача разнообразия (Эшби);
  3. мера сложности структур (Моль);
  4. величина обратно пропорциональная беспорядку в системе;
  5. отраженное разнообразие, то есть нарушение однообразия;
  6. сообщение о чем-либо, уменьшаемая неопределенность, отражение разнообразия в любых объектах и процессах;
- Некоторая совокупность сведений, знаний, которые актуализируемы (получаемы, передаваемы, преобразуемы, сжимаемы и/или регистрируемы) с помощью некоторых знаков (символьного, образного, жестового, звукового, сенсомоторного типа).
- Сведения об окружающем мире и протекающих в нём процессах, воспринимаемые человеком или специальным устройством и передаваемые другим людям устным, письменным или другим способом
- Содержание, полученное от внешнего мира в процессе приспособления к нему (Винер)
- Абстрактное содержание какого-либо высказывания, описания, сообщения или известия
- Сообщения, осведомляющие о положении дел, о состоянии чего-нибудь, обмен сведениями.

Существует одна из формулировок понятия **знание**, которая связывает знание с информацией: **знание** - информация, о которой кто-либо осведомлен. Знание противоположно незнанию - отсутствию достоверной информации о чем-либо.

Количество информации - числовая величина, адекватно характеризующая актуализируемую информацию по разнообразию, сложности, структурированности (упорядоченности), определенности, выбору состояний отображаемой системы.
Если рассматривается некоторая система, которая может принимать одно из n возможных состояний, то актуальной задачей является задача оценки этого выбора, исхода. Такой оценкой может стать мера информации (события).
Мера, как было сказано выше, - непрерывная действительная неотрицательная функция, определенная на множестве событий и являющаяся аддитивной (мера суммы равна сумме мер).
Меры могут быть статические и динамические, в зависимости от того, какую информацию они позволяют оценивать: статическую (не актуализированную; на самом деле оцениваются сообщения без учета ресурсов и формы актуализации) или динамическую (актуализированную т.е. оцениваются также и затраты ресурсов для актуализации информации).

Мера Р. Хартли. Пусть имеется N состояний системы S или N опытов с различными, равновозможными, последовательными состояниями системы. Если каждое состояние системы закодировать, например, двоичными кодами определенной длины d, то эту длину необходимо выбрать так, чтобы число всех различных комбинаций было бы не меньше, чем N. Наименьшее число, при котором это возможно, называется мерой разнообразия множества состояний системы и задается формулой Р. Хартли: $H=k\log_a N$, где k - коэффициент пропорциональности (масштабирования, в зависимости от выбранной единицы измерения меры), a - основание системы меры.
Если измерение ведется в экспоненциальной системе, то k=1, H=lnN (нат); если измерение было произведено в двоичной системе, то k=1/ln2, $H=\log_2 N$ (бит); если измерение было произведено в десятичной системе, то k=1/ln10, H=lgN (дит).
Пример. Чтобы узнать положение точки в системе из двух клеток т.е. получить некоторую информацию, необходимо задать 1 вопрос ("Левая или правая клетка?"). Узнав положение точки, мы увеличиваем суммарную

информацию о системе на 1 бит ($I=\log_2 2$). Для системы из четырех клеток необходимо задать 2 аналогичных вопроса, а информация равна 2 битам ($I=\log_2 4$). Если же система имеет n различных состояний, то максимальное количество информации будет определяться по формуле: $I=\log_2 n$.
Справедливо утверждение Хартли: если в некотором множестве $X=\{x_1, x_2, ..., x_n\}$ необходимо выделить произвольный элемент $x_i \in X$, то для того, чтобы выделить (найти) его, необходимо получить не менее $\log_a n$ (единиц) информации.
Если N - число возможных равновероятных исходов, то величина klnN представляет собой меру нашего незнания о системе.
По Хартли, для того, чтобы мера информации имела практическую ценность, она должна быть такова, чтобы отражать количество информации пропорционально числу выборов.

Мера К. Шеннона. Формула Шеннона дает оценку информации независимо, отвлеченно от ее смысла:

$$I = -\sum_{i=1}^{n} p_i \log_2 p_i$$

где n - число состояний системы; p_i - вероятность (или относительная частота) перехода системы в i-е состояние, причем сумма всех p_i равна 1.
Если все состояния равновероятны (т.е. $p_i=1/n$), то $I=\log_2 n$.
К. Шенноном доказана теорема о единственности меры количества информации. Для случая равномерного закона распределения плотности вероятности мера Шеннона совпадает с мерой Хартли. Справедливость и достаточная универсальность формул Хартли и Шеннона подтверждается и данными нейропсихологии.

Пример. Время t реакции испытуемого на выбор предмета из имеющихся N предметов линейно зависит от $\log_2 N$: $t=200+180\log_2 N$ (мс). По аналогичному закону изменяется и время передачи информации в живом организме. Один из опытов по определению психофизиологических реакций человека состоял в том, что перед испытуемым большое количество раз зажигалась одна из n лампочек, на которую он должен был указать в ходе эксперимента. Оказалось, что среднее время, необходимое для правильного ответа испытуемого, пропорционально не числу n лампочек, а именно величине I, определяемой по формуле Шеннона, где p_i - вероятность зажечь лампочку номер i
Легко видеть, что в общем случае

$$I = -\sum_{i=1}^{n} p_i \log_2 p_i \le \log_2 n$$

Если выбор i-го варианта предопределен заранее (выбора, собственно говоря, нет, $p_i=1$), то I=0.
Сообщение о наступлении события с меньшей вероятностью несет в себе больше информации, чем сообщение о наступлении события с большей вероятностью. Сообщение о наступлении достоверно наступающего события несет в себе нулевую информацию (и это вполне ясно: событие всё равно произойдет когда-либо).

См. также вопрос 2.

2. Знания и ЭВМ.

Аналоговые и непрерывные – АВМ
Цифровые и дискретные – ЭЦВМ, ЭВМ

Понятие информации - одно из основных, ключевых понятий не только в системном анализе, но и в информатике, математике, физике и др. В то же время, это понятие - плохо формализуемое, из-за его всеобщности, объемности, расплывчатости, и трактуется как:

- любая сущность, которая вызывает изменения в некоторой информационно-логической (инфологической - состоящей из сообщений, данных, знаний, абстракций, структурных схем и т.д.) модели, представляющей систему (математика, системный анализ);
- сообщения, полученные системой от внешнего мира в процессе адаптивного управления, приспособления (теория управления, кибернетика);
- отрицание энтропии, отражение меры хаоса в системе (термодинамика);
- связи и отношения, устраняющие неопределенность в системе (теория информации);
- вероятность выбора в системе (теория вероятностей);
- отражение и передача разнообразия в системе (физиология, биокибернетика);
- отражение материи, атрибут сознания, "интеллектуальности" системы (философия).

Мы будем рассматривать системное понимание этой категории, ничуть не отрицая приведенные выше понятия и, более того, используя их по мере надобности.

Процесс познания - это иерархическая система актуализации информации, в которой знания на каждом следующем уровне иерархии являются интегральным результатом актуализации знаний на предыдущем уровне. Это процесс интеграции информационных ресурсов, от получаемых с помощью простого чувственного восприятия и до сложных аксиоматических и абстрактных теорий.

Данные - синтаксические сигналы, образы, актуализируемые с помощью некоторого источника данных. Они рассматриваются безотносительно к семантическому их смыслу.

Информация - это некоторая последовательность сведений, знаний, которые актуализируемы (получаемы, передаваемы, преобразуемы, сжимаемы, регистрируемы) с помощью некоторых знаков символьного, образного, жестового, звукового, сенсомоторного типа.

Информация - это данные, рассматриваемые с учетом некоторой их семантической сущности.

Знания - информация, обеспечивающая достижение некоторой цели и структуры.

Информация с мировоззренческой точки зрения - отражение реального мира. Информация - приращение, развитие, актуализация знаний, возникающее в процессе целеполагающей интеллектуальной деятельности человека.

Никакая информация, никакое знание не появляется сразу: появлению их предшествует этап накопления, систематизации опытных данных, мнений, взглядов, их осмысление и переосмысление. Знание - продукт этого этапа и такого системного процесса.

Информация (в системе, о системе) по отношению к окружающей среде (окружению) бывает трех типов: входная, выходная и внутренняя.

Входная информация - та, которую система воспринимает от окружающей среды. Такого рода информация называется входной информацией (по отношению к системе).

Выходная информация (по отношению к окружающей среде) - та, которую система выдает в окружающую среду.

Внутренняя, внутрисистемная информация (по отношению к данной системе) - та, которая хранится, перерабатывается, используется только внутри системы, актуализируется лишь подсистемами системы.

Внутренние состояния системы и структура системы влияют определяющим образом на взаимоотношения системы с окружающей средой - внутрисистемная информация влияет на входную и выходную информацию, а также на изменение самой внутрисистемной информации.

Основные свойства информации (и сообщений):

- полнота (содержит все необходимое для понимания информации);
- актуальность (необходимость) и значимость (сведений);
- ясность (выразительность сообщений на языке интерпретатора);
- адекватность, точность, корректность интерпретации, приема и передачи;
- интерпретируемость и понятность интерпретатору информации;
- достоверность (отображаемого сообщениям);
- избирательность;
- адресность;
- конфиденциальность;
- информативность и значимость (отображаемых сообщений);
- массовость (применимость ко всем проявлениям);
- кодируемость и экономичность (кодирования, актуализации сообщений);
- сжимаемость и компактность;
- защищенность и помехоустойчивость;
- доступность (интерпретатору, приемнику);
- ценность (предполагает достаточный уровень потребителя).

Охарактеризуем кратко **эмпирико-теоретические методы.**

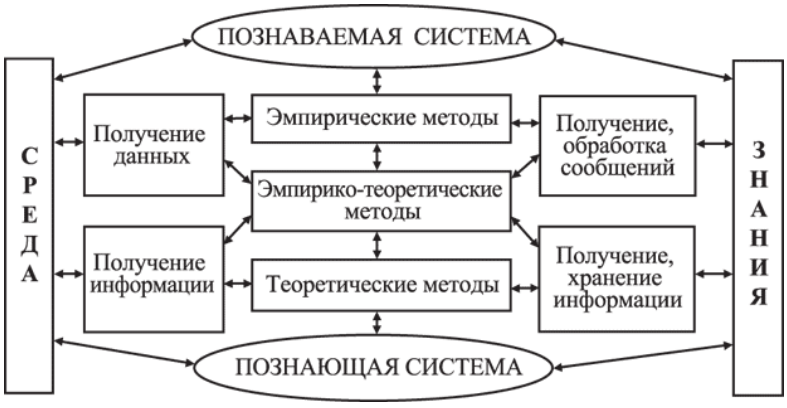
1. Абстрагирование - установление общих свойств и сторон объекта (или объектов), замещение объекта или системы ее моделью. Абстракция в математике понимается в двух смыслах: а) абстракция, абстрагирование - метод исследования некоторых явлений, объектов, позволяющий как выделить основные, наиболее важные для исследования свойства, стороны исследуемого объекта или явления, так и игнорировать несущественные и второстепенные; б) абстракция - описание, представление объекта (явления), получаемое с помощью метода абстрагирования; особо важно в информатике такое понятие как абстракция потенциальной осуществимости, которое позволяет нам исследовать конструктивно объекты, системы с потенциальной осуществимостью (т.е. они могли бы быть осуществимы, если бы не было

ограничений по ресурсам); используются и абстракция актуальной бесконечности (существования бесконечных, неконструктивных множеств, систем и процессов), а также абстракция отождествления (возможности отождествления любых двух одинаковых букв, символов любого алфавита, объектов, независимо от места их появления в словах, конструкциях, хотя их информационная ценность при этом может быть различна).

2. Анализ - разъединение системы на подсистемы с целью выявления их взаимосвязей.
3. Декомпозиция - разъединение системы на подсистемы с сохранением их взаимосвязей с окружением.
4. Синтез - соединение подсистем в систему с целью выявления их взаимосвязей.
5. Композиция - соединение подсистем в систему с сохранением их взаимосвязей с окружением.
6. Индукция - получение знания о системе по знаниям о подсистемах; индуктивное мышление: распознавание эффективных решений, ситуаций и затем проблем, которые оно может разрешать.
7. Дедукция - получение знания о подсистемах по знаниям о системе; дедуктивное мышление: определение проблемы и затем поиск ситуации, его разрешающей.
8. Эвристики, использование эвристических процедур - получение знания о системе по знаниям о подсистемах системы и наблюдениям, опыту.
9. Моделирование (простое моделирование) и/или использование приборов - получение знания об объекте с помощью модели и/или приборов; моделирование основывается на возможности выделять, описывать и изучать наиболее важные факторы и игнорировать при формальном рассмотрении второстепенные.
10. Исторический метод - поиск знаний о системе путем использования ее предыстории, реально существовавшей или же мыслимой, возможной (виртуальной).
11. Логический метод - метод поиска знаний о системе путем воспроизведения ее некоторых подсистем, связей или элементов в мышлении, в сознании.
12. Макетирование - получение информации по макету объекта или системы, т.е. с помощью представления структурных, функциональных, организационных и технологических подсистем в упрощенном виде, сохраняющем информацию, которая необходима для понимания взаимодействий и связей этих подсистем.
13. Актуализация - получение информации с помощью активизации, инициализации смысла, т.е. переводом из статического (неактуального) состояния в динамическое (актуальное) состояние; при этом все необходимые связи и отношения (открытой) системы с внешней средой должны быть учтены (именно они актуализируют систему).
14. Визуализация - получение информации с помощью наглядного или визуального представления состояний актуализированной системы; визуализация предполагает возможность выполнения в системе операции типа "передвинуть", "повернуть", "укрупнить", "уменьшить", "удалить", "добавить" и т.д. (как по отношению к отдельным элементам, так и к подсистемам системы). Это метод визуального восприятия информации.

Охарактеризуем кратко **теоретические методы.**

1. Восхождение от абстрактного к конкретному - получение знаний о системе на основе знаний о ее абстрактных проявлениях в сознании, в мышлении.
2. Идеализация - получение знаний о системе или о ее подсистемах путем мысленного конструирования, представления в мышлении систем и/или подсистем, не существующих в действительности.
3. Формализация - получение знаний о системе с помощью знаков или же формул, т.е. языков искусственного происхождения, например, языка математики (или математическое, формальное описание, представление).
4. Аксиоматизация - получение знаний о системе или процессе с помощью некоторых, специально для этого сформулированных аксиом и правил вывода из этой системы аксиом.
5. Виртуализация - получение знаний о системе созданием особой среды, обстановки, ситуации (в которую помещается исследуемая система и/или ее исследующий субъект), которую реально, без этой среды, невозможно реализовать и получить соответствующие знания.



Зна́ние — в широком смысле совокупность понятий, теоретических построений и представлений.

Зна́ние — форма существования и систематизации результатов познавательной деятельности человека. Знание относится к категории веры, но в отличие от чистой веры, подтверждается опытом или практикой, соответствием ожидаемого и практических результатов.

Зна́ние — субъективный образ объективной реальности, то есть отражение внешнего мира в формах деятельности человека, в формах его сознания и воли.

Зна́ние (предмета) — уверенное понимание предмета, умение самостоятельно обращаться с ним, разбираться в нём, а также использовать для определенных целей.

Зна́ние — информация, о которой кто-либо осведомлен.

Зна́ние -это закономерности предметной области (принципы, связи, законы), полученные в результате практической деятельности и профессионального опыта, позволяющие специалистам ставить и решать задачи в этой области.[Гаврилова]

Зна́ние — проверенный практикой результат познания действительности, верное её отражение в сознании человека. Знание противоположно незнанию, т. е. отсутствию проверенной информации о чём-либо.

3. Критерий, положенный в основу эволюционной классификации ЭВМ.

Технология производства – основной критерий.
Первое поколение ЭВМ /1946-1957гг/ - электронные лампы
Быстродействие их не превышало 2-3 т. оп./сек; емкость ОЗУ - 2-4 К слов. Это ЭВМ: БЭСМ-1 (В.А. Мельников,1955г.), Минск-1 (И.С. Брук 1952/59 гг.), Урал-4 (Б. И. Рамеев), Стрела (Ю.Я. Базилевский, 1953 г.), М-20 (М.К. Сулим 1860 г.). А.Н. Мямлиным была разработана и несколько лет успешно эксплуатировалась "самая большая в мире ЭВМ этого поколения" - машина Восток. *Программирование для этих машин*: однозадачный, пакетный режим, машинный язык, ассемблер.
В ЭВМ второго поколения /1958-1964гг/ элементной базой служили **транзисторы.** Отечественные: Урал-14,Минск-22,БЭСМ-4,М-220,Мир-2,Наири и БЭСМ-6 (1 млн. оп./сек , 128К), Весна (В.С. Полин, В.К. Левин), М-10 (М.А. Карцев). ПС-2000,ПС-3000, УМШМ, АСВТ, Сетунь. *Программирование:* мультипрограммный режим, языки высокого уровня, библиотеки подпрограмм.
Элементная база **ЭВМ третьего поколения, /1965-1971гг/ интегральные схемы** - логически законченный функциональный блок, выполненный печатным монтажом. Отечественные ЭВМ этого поколения ЭВМ ЕС (Единой Системы):ЕС-1010,ЕС-1020, ЕС-1066 (2 млн. оп./сек , 8192К) и др. *Программирование:* мультипрограммный, диалоговый режимы, ОС, виртуальная память.
ЭВМ четвертого поколения /1972-1977гг/ базируются на **"больших интегральных схемах"(БИС) и "микропроцессорах"**. Отечественные - проект "Эльбрус", ПК. *Программирование:* диалоговые режимы, сетевая архитектура, экспертные системы.
ЭВМ **пятого поколения** /начиная с 1978г/ используют **"сверхбольшие интегральные схемы" (СБИС).** Выполненные по такой технологии процессорные элементы на одном кристалле могут быть основным компонентом различных платформ - серверов: от супер-ЭВМ (вычислительных серверов), до интеллектуальных коммутаторов в файл-серверах.

4. Основоположники отечественной вычислительной техники.

ЭВМ: БЭСМ-1 (В.А. Мельников,1955г.), Минск-1 (И.С. Брук 1952/59 гг.), Урал-4 (Б. И. Рамеев), Стрела (Ю.Я. Базилевский, 1953 г.), М-20 (М.К. Сулим 1860 г.). А.Н. Мямлиным была разработана и несколько лет успешно эксплуатировалась "самая большая в мире ЭВМ этого поколения" - машина Восток.
Весна (В.С. Полин, В.К. Левин), М-10 (М.А. Карцев)
В СССР под руководством А.Н. Мямлина в рамках проекта нейронного компьютера велась разработка вычислительной системы, состоящей из специализированных процессоров: процессоров ввода/вывода, вычислительного, символьного, архивного процессоров. МВС-1000 (В.К. Левин, А.В.Забродин). Под руководством Б.А.Бабаяна проектируется микропроцессор Мерсед-архитектуры. В.С. Бурцев разрабатывает проект суперЭМВ на принципах потоковых машин.
Эволюция отечественного программного обеспечения непосредственно связана с эволюцией архитектуры ЭВМ, первая Программирующая Программа ПП, Интерпретирующая Система- ИС создавались для М-20 (ИПМ). Для ЭВМ этого семейства были реализованы компиляторы с Алгола: ТА-1 (С.С.Лавров), ТФ-2 (М.Р.Шура-Бура), Альфа(А.П.Ершов).
Для БЭСМ-6 создан ряд операционные системы: от Д-68 до ОС ИПМ (Л.Н. Королев, В.П. Иванников, А.Н. Томилин, В.Ф.Тюрин, Н.Н. Говорун, Э.З. Любимский).

Под руководством С.С.Камынина и Э.З. Любимского был реализован проект Алмо: создание машинно-ориентированного языка и на его базе системы мобильных трансляторов.
В.Ф.Турчин предложил функциональный язык Рефал, системы программирования на базе этого языка используются при создании систем символьной обработки и в исследованиях в области мета вычислений.

5. Принципы фон-Нейманновской архитектуры ЭВМ.

Программное управление работой ЭВМ.
Программа состоит из последовательности команд, хранимых в Оперативном Запоминающем Устройстве (ОЗУ); каждая команда задает единичный акт преобразования информации. ЭВМ поочередно выбирает команды программы и выполняет предписанные в них дискретные вычисления. В любой момент времени работы ЭВМ выполняется только одна команда программы.
Принцип условного перехода.
Этот принцип дает возможность перехода в процессе вычислений на тот или иной участок программы в зависимости от промежуточных, получаемых в ходе вычислений результатов. Команда условного перехода могут нарушить последовательный порядок выборки команд программы и указать команду для последующего выполнения – L в случае выполнения условий заданного соотношения. (Команды безусловного перехода нарушает порядок выбора команд всегда).
Принцип хранимой программы
Принцип заключается в том, что команды представляются в числовой форме и хранятся в том же Оперативном Запоминающем Устройстве (ОЗУ), что и исходные данные. ОЗУ – структурно состоит из пронумерованных ячеек. Над программой можно производить арифметические действия, изменяя ее динамически.
ВОПРОС: Как можно использовать для модификации программы команду %: %,A1,A2,A3; которая, передает управление команде, размещенной в ОЗУ по адресу A2,
2. пересылает содержимое слова ОЗУ A1 в A3 (A3=A1).
Ответ: Замена команд другими командами программы, изменение порядка следования команд, и т.д.
Использование двоичной системы счисления для представления информации в ЭВМ.
ВОПРОСЫ: 1. Почему числа - степень двойки предпочтительны для измерения параметров оборудования ЭВМ. 2. Почему нумерация строк ОЗУ начинает с нуля.
Ответы: 1. Потому что они характеризуют, сколько двоичных слов может быть обработано или запомнено оборудованием, то есть характеризуют оборудование именно с точки зрения принципов работы ЭВМ. 2. Потому что строки ОЗУ адресуются смещением относительно начала ОЗУ. Соответственно первая строка имеет нулевое смещение, поэтому они и адресуются с 0.
Структура традиционных ЭВМ
Классические (Von Neumann architecture) ЭВМ имеют следующую структуру:

АЛУ + УУ + <кд.....кд.....кд> + ОЗУ, где

ОЗУ (Оперативное Запоминающего Устройства) - память для хранения программ и данных. Таблица, каждая строка которой содержит команду или данное в двоичной системе счисления
УУ (Устройства Управления), устройство, которое последовательно выбирает команды из ОЗУ, дешифрирует их и организует выполнение заданных операций в **АЛУ.**
<кд.....кд.....кд> последовательность команд и данных, причем данные как читаются из ОЗУ, так и туда же записываются.
Совокупность АЛУ и УУ принято называть процессором (ЦПУ,CPU), резервируя слово ЭВМ (ПЭ) для полного вычислительного комплекса. (По словарю А. Синклера "processor" - блок компьютера, выполняющий вычислительные действия). В современных микропроцессорах, микросхема процессора размещается на одном кристалле (чипе) , это: УУ + АЛУ + набор регистров + кэш память. В приведенной схеме не отражены устройства ввода/вывода информации, массовая память для постоянного хранения информации.
Все современные микропроцессоры имеют фон Нейманновскую архитектуру. Для ускорения вычислений которых предложено ряд параллельных архитектур вычислительных машин, для классификации которых , можно использовать нотацию М. Флинна (M.Flynn). *(См. вопрос 30).*

6. Конвейерная обработка данных.

Пример:
Длительность арифметической операции может быть уменьшена за счет временного перекрытия ее различных фаз, путем конвейеризации вычислительной работы. Для этого механизмы Арифметического Логического Устройства (АЛУ) выполняется по конвейерному принципу. Пусть, работа АЛУ - для сложении данных, разделена на три этапа, на три автономных блока Pi : P1 - выравнивание порядков операндов, P2 - операция сложения мантисс, P3 - нормализация результата, каждый из которых выполняется за один условный такт вычислителя.

A1 = B1+C1
A2 = B2+C2
A3 = B3+C3
A4 = B4+C4

Тогда временная диаграмма работы АЛУ имеет вид:

Устройство	1 такт	2 такт	3 такт	4 такт	5 такт	6 такт
P1	B1+C1	B2+C2	B3+C3	B4+C4	нет работы	нет работы
P2	нет работы	B1+C1	B2+C2	B3+C3	B4+C4	нет работы
P3	нет работы	нет работы	B1+C1	B2+C2	B3+C3	B4+C4

ВОПРОСЫ: 1. За сколько тактов будут выполнены эти вычисления, если АЛУ не конвейеризовано. 2. Сокращает ли конвейеризация время выполнения отдельной операции?

Ответы: 1. За 12 тактов. 2. Нет, конвейер предназначен для уменьшения времени выполнения нескольких последовательных команд.

Оптимальную загрузку конвейерных АЛУ можно получить при работе с регулярными структурами, например, по-элементное сложение векторов. В общем случае, пусть работа операционного блока разбивается на n последовательных частей (стадий, выполняемые за одинаковое время), на которых вычислительные операции выполняются в конвейерном режиме. Тогда, если на выполнение одной операции сложения блоку требуется время T, то на обработку N операций сложения время: Tn = (n + N) * (T / n). Следовательно, если n << N, то ускорение вычислений будет в n раз.

Из-за зависимости по данным конвейеризация может терять эффективность. В таких случаях чаще всего используется оптимизация, заключающаяся в изменении порядка выполнения независимых команд **(внеочередное выполнение команд)**. Методы динамической оптимизации: неупорядоченное выполнение - out-of-order execution, неупорядоченная выдача - out-of-order issue.

Пример:

A1 = B1+C1
A2 = A1+C2
A3 = B3+C3
A4 = B4+C4
8 тактов.
А вместо этого:
A1 = B1+C1
A3 = B3+C3
A4 = B4+C4
A2 = A1+C2
6 тактов.

Время выполнения отдельной скалярной операции на конвейерном вычислителе равно: T = S + K, где K - время работы, за которое конвейер выдает очередной результат, а S - время запуска конвейера, время заполнения конвейера, которое без учета времени подготовки операндов, равно: S = K*(m-1), где m - число ступеней конвейера. Производительность конвейерного вычислителя на скалярных операциях (число результатов, выдаваемых за единицу времени) равна: R = 1/(S + K).

Время выполнения векторной операции на конвейерном вычислителе равно: T = S + K*N, где N - длина вектора. Производительность конвейерного вычислителя при векторной работе (число результатов, выдаваемых за единицу времени) равна: R = N/(S + K*N), асимптотическая производительность Rб = 1/K.

7. Зацепление конвейеров.

См. Вопрос 8.

8. Векторно-конвейерные вычислители.

Реализация команд организации цикла (счетчик и переход) при регулярной работе с данными - накладные расходы и препятствие опережающему просмотру на обычных, скалярных вычислителях, показывает, что такие вычисления эффективнее выполнять на специализированном векторно-конвейерном вычислителе.

Если вместо цикла:

DO L=1,N A(I) = B(I)+C(I) ENDDO

использовать запись алгоритма в виде векторной команды сложения вида:

VADD(B,C,A,N), то векторный вычислитель, выполняющий такие команды, будет вырабатывать результаты на каждом такте. В таком вычислителе имеется один (или небольшое число) конвейерный процессор, выполняющий векторные команды путем засылки элементов векторов в конвейер с интервалом, равным длительности прохождения одной стадии обработки. Скорость вычислений зависит только от длительности стадии и не зависит от задержек в процессоре в целом.

Так как конвейер для однотипных операций дешевле и быстрее чем для многофункциональных операций, то выгодно их делать специализированными - однофункциональными: например, только для + или только для *. Для их совместного работы используется принцип **зацепления конвейеров.** Так, в ЭВМ Крей-1 имеется 12 конвейеров, из них 8 могут быть зацеплены, то есть результаты вычисления конвейера могут быть входными аргументами для другого. Операнды (результаты) находятся в памяти верхнего уровня или на регистрах. Для операндов задается: базовый адрес вектора, число элементов, тип данных в каждом элементе, схема хранения вектора в памяти. Некоторые векторные машины могут работать с двух-трех мерными массивами.

9. CISC и RISC архитектуры ЭВМ.

Традиционные процессоры универсальных ЭВМ называются CISC (Complex (Complicated) Instruction Set Computer) - компьютерами с полной системой команд (Семейства Intel ЭВМ от серии ix86 до Pentium и Pentium Pro). Помимо арифметических и логических операций в систему команд ((аппаратно реализуемые функции) включались сложные операции, такие как извлечения корня). Расширение спектра операций облегчает программирование и отладку программ, однако увеличивает трудоемкость разработки процессоров. Время выполнения даже таких операций, как умножение и деление чисел с плавающей запятой варьируется в зависимости от значений аргументов операций, что ограничивает возможности аппаратного планирования совмещенного выполнения команд. Исполнение процессоров на одном кристалле потребовало ревизии системы команд, CISC процессоры не помещались на одном кристалле – чипе СБИС, что привело к реализации компьютеров с сокращенным набором команд - RISC (Reduced Inctruction Set Computing) архитектуры. Традиционное число команд здесь - 128. Выполнение команд, не входящие в этот состав производится программно, для этого в архитектуре предусматривается развитый механизм реализации подпрограмм: реализация запоминания-восстановления регистров, стеки и т.д. Упрощение оборудования, конвейеризация выполнения команд позволило RISC процессорам резко повысить производительность. Недостатки данной архитектуры: отсутствие операций регистр-память вызывают необходимость подкачки данных командами обмена между регистровым файлом процессора и оперативной памятью, а ограниченный формат команд не позволяют минимизировать объем исполняемого кода. Наиболее известны следующие типовые архитектуры RISC процессоров микропроцессоров(МП): PowerPC, PA-RISC, Alpha, SPARC (Scalable Processor Architecture) - МП с изменяемой архитектурой, разработанный в 80 годы в Калифорнийском университете (Беркли); MIPS (Mikroprocessor without Interlocked Pipeline Stages) - МП без блокировки конвейера, разработанный в 80 годы в Стенфордском университете, архитектура запатентована в 1990г.

10. Внеочередное и спекулятивное выполнения команд.

Внеочередное выполнение команд – см. *Вопрос 6.*
В конвейерных архитектурах устройство выборки команд является таким же конвейером, как и другие функциональные устройства. Так, для условного операторы: IF (A<B) GO TO L:S1;L:S2 еще до вычисления значения условного выражения A<B необходимо решать задачу о заполнении конвейера команд кодами S1 или S2 – **спекулятивного выполнения программы** (чтобы не было пропуска тактов конвейера из-за неверно выбранной ветки, коды которой потребуется убирать из конвейера). Про механизмы предсказания переходов *см. Вопрос 11.*

11. Механизмы предсказания переходов.

Проблема: при выполнении переходов (условных) надо заранее поместить в конвейер то, что будет выполняться далее. Тривиальное решение состоит в выборе кода, текстуально следующего за командой условного перехода. Для такого оборудования компиляторы могут формировать объектный код с размещением наиболее вероятно выполняемым фрагменте программы непосредственно за командой условного перехода. Так, для циклических конструкций, вероятность перехода на повторение цикла выше вероятности выхода из него. Некоторые системы программирования дают возможность программисту указывать вероятность перехода по метке в условном переходе.
Аппаратный механизм учета вероятности перехода состоит из блока предсказания переходов. Этот блок, кроме (вместо) **статически** определенных предпочтений для ветвлений, имеет таблицу переходов, в которой хранится история переходов для каждого (в рамках объема таблицы) перехода программы. Большинства современных микропроцессоров обещают точность предсказаний переходов этим способом выше 90%. Причина повышенного внимания к этому вопросу обусловлена большими задержками, возникающими при неверном предсказании переходов, что грозит существенной потерей производительности. Используемые в микропроцессорах методы предсказания переходов, как уже было сказано, бывают **статические и динамические.** Как динамический, так и статический подходы имеют свои преимущества и недостатки.
Статические методы предсказания используются реже. Такие предсказания делаются компилятором еще до момента выполнения программы. Соответствующие действия компилятора можно считать оптимизацией программ. Такая оптимизация может основываться на сборе информации, получаемой при тестовом прогоне программы (Profile Based Optimisation, PBO) или на эвристических оценках. Результатом деятельности компилятора являются "советы о направлении перехода", помещаемые непосредственно в коды выполняемой программы. Эти советы использует затем аппаратура во время выполнения. В случае, когда переход происходит, или наоборот - как правило, не происходит, советы компилятора часто бывают весьма точны, что ведет к отличным результатам. Преимущество статического подхода - отсутствие необходимости интегрировать на чипе дополнительную аппаратуру предсказания переходов.
Большинство производителей современных микропроцессоров снабжают их различными средствами **динамического предсказания** переходов, производимого на базе анализа "предыстории". Тогда аппаратура собирает статистику переходов, которая помещается в таблицу истории переходов BHT (Branch History Table).В обоих случаях компилятор не может выработать эффективные рекомендации на этапе трансляции программы. В то же время схемы динамического предсказания переходов легко справляются с такими задачами. В этом смысле

динамическое предсказание переходов "мощнее" статического. Однако у динамического предсказания есть и свои слабые места - это проблемы, возникающие из-за ограниченности ресурсов для сбора статистики.

12. Управление виртуальной памятью.

Адресация памяти производится с точностью до байта, длина адреса, его разрядность, определяет пространство памяти, которое может быть доступно ("видимо") в программе. Так 32 разрядный **виртуальный адрес** охватывает пространство в 4 Гбайта. Это **виртуальное пространство** - математическая память программы может не совпадать с реальным, **физическим пространством памяти ЭВМ.**

Адреса виртуального пространства памяти - виртуальные адреса, а адреса физического пространства - физические адреса.

Механизм виртуальной памяти позволяет:

- снять ограничения, связанные с объемом памяти, при разработки алгоритмов;
- предоставлять программисту область памяти в виде логически непрерывного пространства;
- способствовать более эффективному управлению физической памятью.

Процесс преобразование виртуальных адресов в физические при выполнении программы называется трансляцией адресов, наиболее распространенный механизм для этого - страничная память. Механизмы виртуальной памяти реализуется путем разбиения памяти и виртуальной и физической на одинаковые страницы, обычно, размером 4 Кбайта. Адрес разделяется на две части в соответствии с принятой длиной страницы: номер страницы (а) и адрес внутри страницы - сдвиг, смещение (б). Трансляция адреса

Виртуальный адрес Физический адрес

а б -> с б

Аппаратно трансляция адресов производится при помощи таблицы страниц. Каждой странице виртуальной памяти соответствует строка в таблице страниц, объем которой соответствует числу страниц виртуальной памяти. В i строке таблицы хранится: N страницы (блока) физической памяти, которая соответствует данной виртуальной, статус доступа (чтение, запись), признак записи. Полный физический адрес получается добавлением к физическому адресу, полученного из таблице страниц, смещения внутри страницы (б). Реальная структура таблицы страниц имеет более сложный вид.

13. Ассоциативная память.

Оперативную память (ОП) можно представить в виде двумерной таблицы, строки которой хранят в двоичном коде команды и данные. Обращения за содержимом строки производится заданием номера (адреса) нужной строки. При записи, кроме адреса строки указывается регистр, содержимое которого следует записать в эту строку. Запись занимает больше времени, чем чтение. Пусть в памяти из трех строк хранятся номера телефонов.

1924021
9448304
3336167

Для получения номера телефона второго абонента следует обратиться: load (2) и получить в регистре ответа R = 9448304. Такой вид памяти, при котором необходимая информация определяется номером строки памяти, называется адресной. Другой вид оперативной памяти – ассоциативной можно рассматривать также как двумерную таблицу, но у каждой строки которой есть дополнительное поле, поле ключа. Например:

Ключ	Содержимое
Иванов	1924021
Петров	9448304
Сидоров	3336167

После обращение к ассоциативной памяти с запросом : load (Петров) для данного примера получим ответ: R = 9448304. Здесь задание координаты строки памяти производится не по адресу, а по ключу. Но при состоянии ассоциативной памяти:

Ключ	Содержимое
1	1924021
2	9448304
3	3336167

можно получить номер телефона из второй строки запросом: load (2). Таким образом на ассоциативной памяти можно моделировать работу адресной. Ассоциативная память имеет очевидное преимущества перед адресной, однако, у нее есть большой недостаток - ее аппаратная реализация невозможна для памяти большого объема.

ВОПРОС: Предложите схему реализации модели ассоциативной памяти при помощи адресной.

Ответ: Область памяти делим ровно пополам. Первая половина заполняется ключами, вторая соответствующими ключам значениями. Когда найден ключ, известен его адрес как смещение относительно начала памяти. Тогда адрес содержимого по ключу – это смещение + размер области ключей, то есть адрес ячейки из второй половины памяти, которая соответствует ключу.

14. Назначение и структура кэш-памяти.

Для ускорения доступа к оперативной памяти используется буферизация данных и объектного кода в памяти, скорость которой значительно выше основной. Если бы доступ к любым типам данных был случайным, то буферизация была бы бесполезным. Эффект от буферизации можно определить среднему времени выборки: t = t2*p+ t1*(1-p), где t1 - среднее время доступа к данным основной памяти, t2 - среднее время доступа к данным из буфера (t2<t1), p - вероятность наличия данного в буфере. Очевидно, среднее время зависит от вероятности p и изменяется от среднего времени доступа к основной памяти (при p=0) до среднего времени доступа непосредственно в буфер (при p=1). Кэш (cache, cache memory) память, как правило, на порядок более быстрая, чем основная, размещается в качестве буферной, между процессором и основной памятью и служит для временного хранения (в рамках своего объема) всех данных, потребляемых или генерирумых процессором. В много-уровневых кэшах элементами связки: "процессор - основная память" могут выступать сами кеши. Алгоритм кэширования состоит в следующем:

1. По каждому запросу процессора происходит поиск требуемого данного в кэш памяти (места для записи генерируемого данного).
2. Если данное (место) есть в кэше - кэш попадание (cache hit), то оно передается в процессор (из процессора).
3. Если нужного данного нет в кэше - кэш промах (cache miss), данное из основной памяти пересылается в кэш память, и передаются также процессору. При переполнении кэша (нет места для записи), из него удаляются (модифицированные данные сохраняются в основной памяти) часть данных, обычно, наименее востребованные.

Традиционным кэшем является "процессорный" кэш или кэш первого уровня (первичный) или кэш (L1 Cache), имеющийся на любом микропроцессоре. Это буферная память объемом от 4 Кбайт до 16 Мбайт, в которой размещаются все данные, адресуемые процессором, и из которой данные поставляются процессору. Эта память значительно быстрее основной, но меньшего объема, поэтому механизм кэширования обеспечивает обновление кэша, обычно, сохраняя в нем только наиболее часто употребляемые данные. Обмен между основной и кэш памятями производится квантами, объемами 4 - 128 байт - копируются "строки кэша" (cache line), содержащие адресуемое данное.

Обычно, программный код кешируется через особый, I кэш память, отделенной от кэша данных D-кэша. Выборка данных из кэша (hit time) прозводится, обычно, за один такт синхронизации (оценки 1 - 4 такта), потери при кеш промахе оцениваются в 8 - 32 такта синхронизации, доля промахов (miss rate) - 1% -20%. По определению, эффект кэширования основан на предположении о многократном использовании данных (Data reuse) из кэш памяти. Принято различать две формы многократного использования данных кэша:- временное использование (temporal reuse). - пространственное использование (spatial reuse). Временное использование означает, что некоторое данное, загруженные в кэш, может использоваться, по крайней мере, более двух раз. Пространственное использование кэша предполагает возможность использовать некоторый пространственный набор данных - строки кэша. Архитектура кэш памяти: полностью ассоциативная, частично ассоциативная и кэш память с прямым отображением.

Для согласования содержимого кэш-памяти и оперативной памяти используют три метода записи:

- Сквозная запись (write through) - одновременно с кэш-памятью обновляется оперативная память.
- Буферизованная сквозная запись (buffered write through) - информация задерживается в кэш-буфере перед записью в оперативную память и переписывается в оперативную память в те циклы, когда ЦП к ней не обращается.
- Обратная запись (write back) - используется бит изменения в поле тега, и строка переписывается в оперативную память только в том случае, если бит изменения равен 1.

Как правило, все методы записи, кроме сквозной, позволяют для увеличения производительности откладывать и группировать операции записи в оперативную память.

В структуре кэш-памяти выделяют два типа блоков данных:

- память отображения данных (собственно сами данные, дублированные из оперативной памяти);
- память тегов (признаки, указывающие на расположение кэшированных данных в оперативной памяти).

Пространство памяти отображения данных в кэше разбивается на строки - блоки фиксированной длины (например, 32, 64 или 128 байт). Каждая строка кэша может содержать непрерывный выровненный блок байт из оперативной памяти. Какой именно блок оперативной памяти отображен на данную строку кэша, определяется тегом строки и алгоритмом отображения. По алгоритмам отображения оперативной памяти в кэш выделяют три типа кэш-памяти:

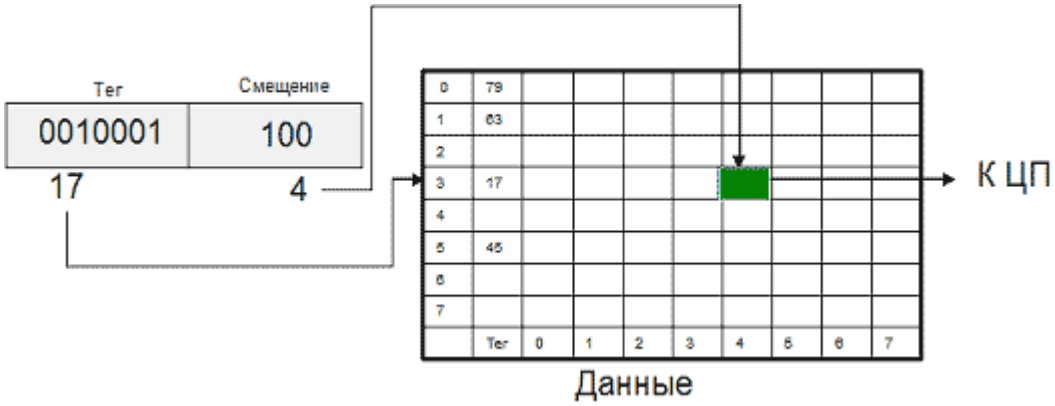
- полностью ассоциативный кэш;
- кэш прямого отображения;
- множественный ассоциативный кэш.

15. Полностью ассоциативная кэш-память.

http://www.intuit.ru/department/hardware/csorg/9/2.html

Для полностью ассоциативного кэша характерно, что кэш-контроллер может поместить любой блок оперативной памяти в любую строку кэш-памяти (рис.). В этом случае физический адрес разбивается на две части: смещение в блоке (строке кэша) и номер блока. При помещении блока в кэш номер блока сохраняется в теге соответствующей строки. Когда ЦП обращается к кэшу за необходимым блоком, кэш-промах будет обнаружен только после сравнения тегов всех строк с номером блока.

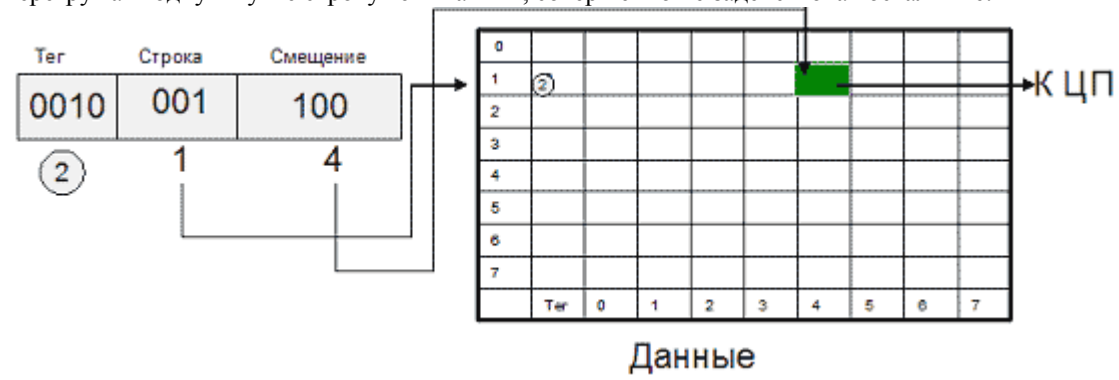
Одно из основных достоинств данного способа отображения - хорошая утилизация оперативной памяти, т.к. нет ограничений на то, какой блок может быть отображен на ту или иную строку кэш-памяти. К недостаткам следует отнести сложную аппаратную реализацию этого способа, требующую большого количества схемотехники (в основном компараторов), что приводит к увеличению времени доступа к такому кэшу и увеличению его стоимости.



16. Кэш-память с прямым отображением.

Альтернативный способ отображения оперативной памяти в кэш - это кэш прямого отображения (или одновходовый ассоциативный кэш). В этом случае адрес памяти (номер блока) однозначно определяет строку кэша, в которую будет помещен данный блок. Физический адрес разбивается на три части: смещение в блоке (строке кэша), номер строки кэша и тег. Тот или иной блок будет всегда помещаться в строго определенную строку кэша, при необходимости заменяя собой хранящийся там другой блок. Когда ЦП обращается к кэшу за необходимым блоком, для определения удачного обращения или кэш-промаха достаточно проверить тег лишь одной строки.

Очевидными преимуществами данного алгоритма являются простота и дешевизна реализации. К недостаткам следует отнести низкую эффективность такого кэша из-за вероятных частых перезагрузок строк. Например, при обращении к каждой 64-й ячейке памяти в системе на рис. 9.2 кэш-контроллер будет вынужден постоянно перегружать одну и ту же строку кэш-памяти, совершенно не задействовав остальные.



Пример:

Если объем ОЗУ – 4 Гбайт, тогда полный адрес - 32 бита можно представить в виде полей: 20 рр. – тэг (Т), 7 рр – номер строки таблиц кэша (S), 5 рр – номер байта в строке (N). Поиск запрошенного байта (T-S-N) в кэше с прямым распределением производится так:

1. Из памяти данных и памяти тэгов кэша одновременно считываются S-ные строки.
2. Если содержимое считанной строки памяти тэгов равно Т – кэш попадание, это значит, что считанная S строка памяти данных кэша содержит запрашиваемый байт и его номер в строке есть N.
3. Если содержимое считанной строки памяти тэгов не равно Т – кэш промах, и тогда T-S строка ОЗУ переписывается в S строку памяти данных кэша, а Т записывается в S строку памяти тэгов. Затем, см по п. 1.

17. Частично ассоциативная кэш-память.

См. также Вопросы 15, 16. Компромиссным вариантом между первыми двумя алгоритмами является множественный ассоциативный кэш или частично-ассоциативный кэш (рис.). При этом способе организации кэш-памяти строки объединяются в группы, в которые могут входить 2, 4, 8, и т.д. строк. В соответствии с количеством строк в таких группах различают 2-входовый, 4-входовый и т.п. ассоциативный кэш. При обращении к памяти физический адрес разбивается на три части: смещение в блоке (строке кэша), номер группы (набора) и тег. Блок памяти, адрес которого соответствует определенной группе, может быть размещен в любой строке этой группы, и в теге строки размещается соответствующее значение. Очевидно, что в рамках выбранной группы соблюдается

принцип ассоциативности. С другой стороны, тот или иной блок может попасть только в строго определенную группу, что перекликается с принципом организации кэша прямого отображения. Для того чтобы процессор смог идентифицировать кэш-промах, ему надо будет проверить теги лишь одной группы (2/4/8/… строк).

18. Дисциплина обновления кэш-памяти.

Стратегии обновления данных в кэш смотрите в *Вопросе 19.* Далее рассматриваются стратегии обновления данных в памяти по обновления данных процессором.

WriteBack

В схеме обновления с обратной записью используется бит "изменения" в поле тэга. Этот бит устанавливается, если блок был обновлен новыми данными и является более поздним, чем его оригинальная копия в основной памяти. Перед тем как записать блок из основной памяти в кэш-память, контроллер проверяет состояние этого бита. Если он установлен, то контроллер переписывает данный блок в основную память перед загрузкой новых данных в кэш-память (то есть только тогда, когда уже не нужны обновленные данные в кэш и они замещаются).

Обратная запись быстрее сквозной, так как обычно число случаев, когда блок изменяется и должен быть переписан в основную память, меньше числа случаев, когда эти блоки считываются и перезаписываются.

Однако обратная запись имеет несколько **недостатков.** Во-первых, все измененные блоки должны быть переписаны в основную память перед тем, как другое устройство сможет получить к ним доступ. Во-вторых, в случае катастрофического отказа, например, отключения питания, когда содержимое кэш-памяти теряется, но содержимое основной памяти сохраняется, нельзя определить, какие места в основной памяти содержат устаревшие данные. Наконец, контроллер кэш-памяти для обратной записи содержит больше (и более сложных) логических микросхем, чем контроллер для сквозной записи. Например, когда система с обратной записью осуществляет запись измененного блока в память, то она формирует адрес записи из тэга и выполняет цикл обратной записи точно так же, как и вновь запрашиваемый доступ.

WriteThru

Сквозная запись.

При обновлении кэш-памяти методом сквозной записи контроллер кэш-памяти одновременно обновляет содержимое основной памяти. Иначе говоря, основная память отражает текущее содержимое кэш-памяти. Быстрое обновление позволяет перезаписывать любой блок в кэш-памяти в любое время без потери данных. Система со сквозной записью проста, но время, требуемое для записи в основную память, снижает производительность и увеличивает количество обращений по шине (что особенно заметно с мультизадачной системе).

Буферизованная сквозная запись.

С схеме обновления с буферизованной сквозной записью любая запись в основною память буферизуется, то есть информация задерживается в кэш-памяти перед записью в основную память (схемы кэш-памяти управляют доступом к основной памяти асинхронно по отношению к работе процессора). Затем процессор начинает новый цикл до завершения цикла записи в основную память. Если за записью следует чтение, то это кэш-попадание, так как чтение может быть выполнено в то время, когда контроллер кэш-памяти занят обновлением основной памяти. Эта буферизация позволяет избежать снижения производительности, характерного для системы со сквозной записью (запись производится в тот момент, пока процессор читает из кэша и занят чем чем-то кроме обновления данных).

У этого метода есть один существенный недостаток. Так как обычно буферизуется только одиночная запись, то две последовательные записи в основную память требуют цикла ожидания процессора. Кроме этого, запись с пропущенным последующим чтением также требует ожидания процессора. Состояние ожидания - это внутреннее состояние, в которое входит процессор при отсутствии синхронизирующих сигналов. Состояние ожидания используется для синхронизации процессора с медленной памятью.

Запись с размещением и без.

Предыдущие подходы описывают только случаи кэш-попадания. Однако случаи, когда обновляемые данные в КЭШе отсутствуют, также возможны. Тогда данные пишутся в ОЗУ и потом копируются в кэш. Запись без рахмещения – данные не копируются в кэш. Обычно при использовании стратегии WriteThru размещение не делается, а при использовании обратной записи делается – есть надежда, что не придется снова лезть в память для записи в след. раз.

19. Стратегии записи в кэш-память.

Стратегии обновления данных в памяти рассмотрены в Вопросе 18. Далее рассматривается случай записи новых данных из памяти в кэш при кэш-промахе.
Для случая прямого отображения стратегия тривиальна – замещается строка, в которой может располагаться данных блок ОЗУ. Для ассоциативной организации кэша (полностью или частично) надо выбирать, какую из строк замещать новыми данными. Две стратегии: случайная или LRU (Least Recently Used) – заменяется та, которую дольше всех не использовали. Сложность – надо фиксировать все обращения к строкам кэша, чтобы вычислять наиболее неиспользуемую строку. Стоит отметить, что доли промахов с ростом кэша для случайного алгоритма уменьшаются быстрее, так что эффективность применения LRU снижается.

20. Расслоение памяти.

Реализация оперативной памяти с использованием некоторого множества микросхем позволяет использовать заложенный в ней потенциальный параллелизм. Для этого микросхемы памяти объединяются в банки или модули, содержащие фиксированное число слов, причем только к одному из этих слов банка возможно обращение в каждый момент времени. Чтобы получить большую скорость доступа, нужно осуществлять одновременный доступ к нескольким банкам памяти. Одна из общих методик, используемых для этого, называется расслоением памяти. При расслоении памяти банки организуются так, чтобы N последовательных адресов памяти i, i + 1, i + 2, ..., i + N-1 приходились на N различных банков. Степень или коэффициент расслоения определяют распределение адресов по банкам памяти. Схема управления памятью реализует конвейер совмещения обращений к различным блокам памяти. Такая организация памяти увеличивает в N раз обращения по последовательным адресам, что является характерным при загрузке информации в кэш память и сохранении одного из её блоков. При подобной организации оперативной памяти можно использовать для неё микросхемы в N раз более медленные, чем микросхемы кэш памяти и не увеличивать разрядность шины данных.
Наиболее общим случаем расслоения памяти является возможность реализации нескольких независимых обращений, когда несколько контроллеров памяти позволяют банкам памяти работать самостоятельно. Такое решение наиболее характерно для многопроцессорных компьютеров.
Для банков одинаковой емкости: B1,B2,B3,..Bm-1 адрес i трансформируется в адрес d внутри банка Bb расчетом:
i=d * m + b, где d=>0, 0<=b<=m-1
При расслоении на четыре распределение адресов в банках будет:

Адреса в банках-b	Банк 1	Банк 2	Банк 3	Банк 4
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

21. Принципы VLIW архитектуры.

В ЭВМ с архитектурой VLIW (Very Long Instruction Word) - (очень длинные командные слова), команды могут иметь широкий формат (длину) и команда может содержать несколько содержательных инструкций, выполнение которых детально регламентируется в терминах тактов работы АЛУ (параллельное выполнение нескольких команд в АЛУ). В таких архитектурах имеется возможность программировать вычислительные алгоритмы (включая векторные) с максимальной производительностью для данной аппаратуры. В них вся работа по оптимальному программированию возлагается на системы программирования (или ручное программирование).
Однако упрощения в архитектуре управления приводит к значительному возрастанию сложности задачи планирования выдачи команд, так программными средствами должна быть обеспечена точная синхронизация считывания и записи данных. При этом необходимо так планировать параллельное выполнение операций машины, чтобы выполнялись определенные ограничения на число одновременно считываний и записей в наборы регистров, использование ФУ и т.д. Размер командного слова в машинах данной архитектуры - FPS (AP-120B) - 64 бита, Multilow Tract - 1024.
Определяющие свойства архитектуры VLIW:
- Одно центральное управляющее устройство (УУ), обрабатывающее за один такт одну длинную команду.
- Большое число функциональных устройств (ФУ) - АЛУ.
- Наличия в длинной команде полей, каждое из которых содержит команду управления некоторым функциональным устройством или команду обращения к памяти.
- Статически определенная длительность в тактах исполнения каждой операции. Операции могут быть конвейеризованы**.**
- Закрепление во время компиляции банков расслоенной памяти за ФУ для получения максимальной ширины доступа для данных, которые можно соединить в одну команду.
- Система передвижения данных между ФУ минуя память. Маршрут передвижения полностью специфицируется во время компиляции.
- Практическая невозможность ручного программирования в силу большой сложности возникающих комбинаторных задач. То есть требуются специальные системы программирования и оптимизации. В отличие от суперскалярных процессоров, здесь требуется статическая (на этапе компиляции) распараллеливание.

22. Суперскалярные и мультитредовые архитектуры микропроцессоров.

Суперскалярный процессор представляет собой нечто большее, чем обычный последовательный (скалярный) процессор. В отличие от последнего, он может выполнять несколько операций за один такт. Основными компонентами суперскалярного процессора являются устройства для интерпретации команд (УУ), снабженные логикой, позволяющей определить, являются ли команды независимыми, и достаточное число исполняющих устройств (ФУ, АЛУ). В исполняющих устройствах могут быть конвейеры. Суперскалярные процессоры реализуют параллелизм на уровне команд. Примером компьютера с суперскалярным процессором является IBM RISC/6000. Тактовая частота процессора у ЭВМ была 62.5 МГц, а быстродействие системы на вычислительных тестах достигало 104 Mflop (Mflop - единица измерения быстродействия процессора - миллион операций с плавающей точкой в секунду). Суперскалярный процессор не требует специальных векторизующих компиляторов, хотя компилятор должен в этом случае учитывать особенности архитектуры. Итак, суперскалярные процессор призван, в отличие от VLIW, динамически определять места распараллеливания.

Другой, по сравнению с организацией кэш-памяти, метод построения внутрикристальной памяти применяется в **мультитредовой** архитектуре, основная особенность которой – использование совокупности регистровых файлов (добавление УУ при одном АЛУ). Эта архитектура решает проблему разрыва между скоростью обработки в процессоре и временем доступа в основную память за счет переключения в каждом такте процессора на работу с очередным регистровым файлом. Каждый регистровый файл обслуживает один вычислительный процесс – тред (поток). Всего в каждом процессоре имеется n регистровых файлов, поэтому запрос, выданный в основную память каждым из потоков, может обслуживаться в течение n-1 такта, вплоть до момента, когда процессор снова переключится на тот же регистровый файл. Выбор значения n определяется отношением времени доступа в память ко времени выполнения команды процессором. Конечно, задача формирования потоков из последовательной программы должна, по возможности, решаться компилятором. В противном случае будущее этой архитектуры окажется ограниченным узкой проблемной ориентацией.

Компания Tera объявила о разработке проекта мультитредового микропроцессора, реализующего процессор MTA. Level One, приобретенная Intel, выпустила мультитредовый сетевой микропроцессор IXP1200, содержащий в своем составе 6 четырехтредовых процессоров. IBM анонсировала проект компьютера Blue Gene, кристалл микропроцессора которого включает 32 восьмитредовых процессора. В кристалл встроена память EDRAM, организованная в 32 блока. Каждый блок соответствует одному из 32 процессоров и имеет шину доступа 256 разрядов. Поскольку EDRAM обладает высокой пропускной способностью и малой задержкой, то при восьмитредовой структуре процессора становится возможным отказаться от кэш-памяти, вместо которой между процессором и памятью используется небольшая буферная память.

23. Стандарт IA-64.

Материал: http://www.ixbt.com/cpu/ia64.html

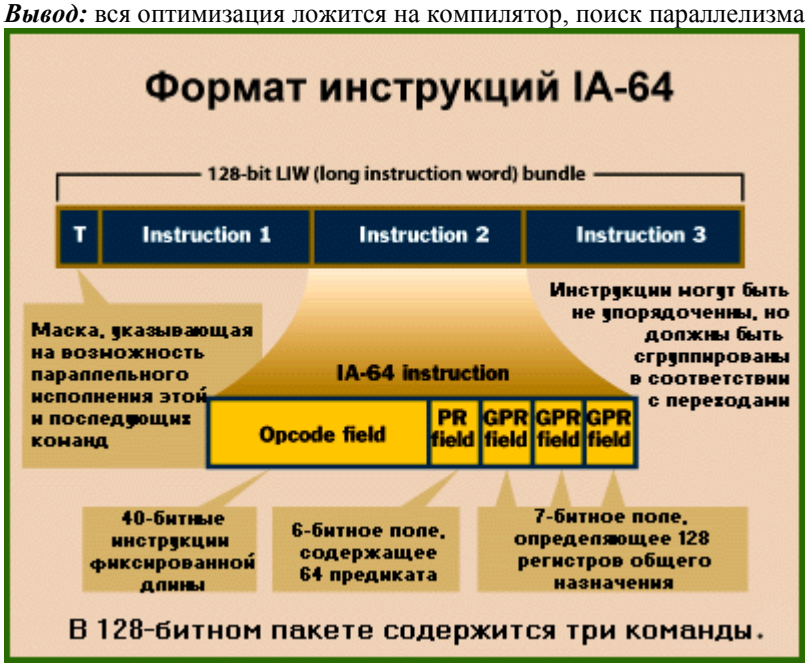
Дополнительно: http://joyous-life.ru/cgi-bin/index.cgi?&id=44

- Команды в формате IA-64 упакованы по три в 128-битный пакет для быстрейшей обработки.
- Каждый 128-битный пакет содержит шаблон (template) длиной в несколько бит, помещаемый в него компилятором, который указывает процессору, какие из команд могут выполняться параллельно. Теперь процессору не нужно будет анализировать поток команд в процессе выполнения для выявления "скрытого параллелизма". Вместо этого наличие параллелизма определяет компилятор и помещает информацию в код

программы. Каждая команда (как для целочисленных вычислений, так и для вычислений с плавающей точкой) содержит три 7-битных поля регистра общего назначения (РОН). Из этого следует, что процессоры архитектуры IA-64 содержат 128 целочисленных РОН и 128 регистров для вычислений с плавающей точкой. Все они доступны программисту и являются регистрами с произвольным доступом (programmer-visible random-access registers). Шаблон пакета (bundle's template field) указывает не только на то, какие команды в пакете могут выполняться независимо, но и какие команды из следующего пакета могут выполняться параллельно. Команды в пакетах не обязательно должны быть расположены в том же порядке, что и в машинном коде, и могут принадлежать к различным путям ветвления. Компилятор может также помещать в один пакет зависимые и независимые команды, поскольку возможность параллельного выполнения определяется шаблоном пакета. По сравнению с процессорами x86, у которых всего восемь целочисленных РОН и стек глубины 8 для вычислений с плавающей точкой, IA-64 намного "шире" и, соответственно, будет намного реже простаивать из-за "нехватки регистров".

- Компиляторы для IA-64 будут использовать технологию "отмеченных команд" (predication) для устранения потерь производительности из-за неправильно предсказанных переходов и необходимости пропуска участков кода после ветвлений. Когда процессор встречает "отмеченное" ветвление в процессе выполнения программы, он начинает одновременно выполнять все ветви, уникально помечая каждую из них (компилятор помечает каждую команду предикатов ветки, к которой она принадлежит). Максимально – 64 ветви. После того, как будет определена "истинная" ветвь, процессор сохраняет необходимые результаты и сбрасывает остальные.
- Компиляторы для IA-64 будут также просматривать исходный код с целью поиска команд, использующих данные из памяти. Найдя такую команду, они будут добавлять пару команд - команду предварительной загрузки (speculative loading) и проверки загрузки (speculative check). Во время выполнения программы первая из команд загружает данные в память до того, как они понадобятся программе. Вторая команда проверяет, успешно ли произошла загрузка, перед тем, как разрешить программе использовать эти данные. Предварительная загрузка позволяет уменьшить потери производительности из-за задержек при доступе к памяти, а также повысить параллелизм.

Вывод: вся оптимизация ложится на компилятор, поиск параллелизма также возлагается на компилятор.



24. Оптимизация программ под архитектуру микропроцессора.

25. Гетерогенные распределенные вычислительные системы.

Ключевое слово – **ИНТРАНЕТ**.

Мультикомпьютерные распределенные системы разделяют на гомогенные и **гетерогенные**. Для гомогенных систем характерна одна соединяющая компьютеры сеть, использующая единую технологию. Одинаковы также и все процессоры, имеющие одинаковый объем собственной памяти. Гомогенные мультикомпьютерные системы нередко используют в качестве параллельных (работающих с одной задачей), как и мультипроцессорные. В отличие от них гетерогенные мультикомпьютерные системы могут содержать целую гамму независимых компьютеров, соединенные разнообразными сетями. Далее см. *Таненбаум, стр. 44, 1.3.3.* **Есть централизованное управление,** подразумевается наличие распределенной программной системы (ОС, ФС, и т.д.).

26. Метакомпъютинг.

Ключевое слово – **ИНТЕРНЕТ.**

Grid (метакомпьютинг) - растущая инфраструктура, которая в скором будущем видимо фундаментально изменит наше представление о вычислительных сетях и их возможностях. Grid призвана объединить множество региональных и национальных сетей, создав, таким образом, универсальный источник вычислительных ресурсов, доступных широкому кругу пользователей. Авторами идеи являются сотрудник Аргонской Национальной Лаборатории Университета города Чикаго Ян Фостер (Ian Foster) и сотрудник Института Информатики Университета Южной Калифорнии Карл Кессельман (Carl Kesselman). Слово grid (сетка) для названия новой технологии было выбрано по аналогии с термином "электрическая сеть", которая в свое время предоставила всепроникающий доступ к источникам электроэнергии, и, подобно компьютерной технике, оказала огромное воздействие на человеческое общество. Создатели grid предположили, что, обеспечение надежного и недорогого доступа к вычислительным сетям инициирует похожий преобразовательный эффект и стимулирует появление новых классов сетевых компьютерных приложений.

Использование компьютеров для решения актуальных задач давно уже стало естественной средой человеческого существования. Современные вычислительные системы применяются для моделирования сложнейших научных и технических проблем, медицинской диагностики, управления промышленным оборудованием, прогнозирования погоды и многого другого. Тем не менее, интенсивность и эффективность использования компьютерной техники до сих пор мала. Ограниченность использования вычислительных ресурсов объясняется множеством причин, включая нехватку соответствующего образования у пользователей и средств на приобретение достаточно мощной компьютерной техники. Поэтому, практически с момента возникновения компьютеров ставился вопрос об объединении большого количества вычислительных ресурсов. Реальная возможность для такого объединения появилась только в последнее время. Её появление обусловило: развитие компьютерных архитектур и непрерывный рост производительности, улучшение пропускной способности коммуникационных сред, а также новые технологии разработки программ. **Что важно**, метакомпьютер может не иметь постоянной конфигурации - отдельные компоненты могут включаться в его конфигурацию или отключаться от нее; при этом технологии метакомпьютинга обеспечивают непрерывное функционирование системы в целом. Современные исследовательские проекты в этой области направлены на обеспечение прозрачного доступа пользователей через Интернет к необходимым распределенным вычислительным ресурсам, а также прозрачного подключения простаивающих вычислительных систем к метакомпьютерам.

Очевидно, что наилучшим образом для решения на метакомпьютерах подходят задачи **переборного и поискового типа**, где вычислительные узлы практически не взаимодействуют друг с другом и основную часть работы производят в автономном режиме: задача исследования генома, дешифрация, обработка результатов физических испытаний, проект CETI. Основная схема работы в этом случае примерно такая: специальный агент, расположенный на вычислительном узле (компьютере пользователя), определяет факт простоя этого компьютера, соединяется с управляющим узлом метакомпьютера и получает от него очередную порцию работы (область в пространстве перебора). По окончании счета по данной порции вычислительный узел передает обратно отчет о фактически проделанном переборе или сигнал о достижении цели поиска.

Далее на данной странице будут вкратце описаны и приведены ссылки на основные исследовательские проекты в области мета-компьютинга, разработанные программные технологии, конкретные примеры мета-компьютеров. **Нет централизованного управление**, основные архитектуры – Legion (объектно-ориентир), Globus (в терминах предоставления сервисов).

27. Кластерные архитектуры.

Вычислительные кластеры (cluster) – группа ЭВМ (серверов), связанная между собой системной сетью и **функционирующая с точки зрения пользователя как единый вычислительный узел.** Локальные сети отличаются от кластеров тем, что узлы локальной сети используются индивидуально, в соответствии со своим назначением. В свою очередь кластеры разделяются на Высокоскоростные (High Performance, HP) и Системы Высокой Готовности (High Availability, HA), кластеры с балансировкой нагрузки (Load balancing clusters), а также Смешанные Системы. Высокоскоростные системы предназначены для задач, которые требуют больших вычислительных мощностей: обработка изображений, научные исследования, математическое моделирование и т. д.

Кластеры высокой готовности используются в банковских операциях, электронной коммерции и т д.

Другой источник:

Архитектура Набор рабочих станций (или даже ПК) общего назначения, используется в качестве дешевого варианта массивно-параллельного компьютера. Для связи узлов используется одна из стандартных сетевых технологий (Fast/Gigabit Ethernet, Myrinet) на базе шинной архитектуры или коммутатора. При объединении в кластер компьютеров разной мощности или разной архитектуры, говорят о гетерогенных (неоднородных) кластерах.

Узлы кластера могут одновременно использоваться в качестве пользовательских рабочих станций. В случае, когда это не нужно, узлы могут быть существенно облегчены и/или установлены в стойку.

Примеры NT-кластер в NCSA, Beowulf-кластеры.
Операционная система Используются стандартные для рабочих станций ОС, чаще всего, свободно распространяемые - Linux/FreeBSD, вместе со специальными средствами поддержки параллельного программирования и распределения нагрузки.
Модель программирования Программирование, как правило, в рамках модели передачи сообщений (чаще всего - MPI). Дешевизна подобных систем оборачивается большими накладными расходами на взаимодействие параллельных процессов между собой, что сильно сужает потенциальный класс решаемых задач.

28. Симметричные мультипроцессорные системы.

Системы данного класса: SMP (Scalable Parallel Processor) состоят из нескольких однородных процессоров и массива общей памяти (разделяемой памяти – shared memory): любой процессор может обращаться к любому элементу памяти. По этой схеме построены 2,4 процессорные SMP сервера на базе процессоров Intel, HP и т. д., причем процессоры подключены к памяти с помощью общей шины. Системы с большим числом процессоров (но не более 32) подключаются к общей памяти, разделенной на блоки, через не блокирующийся полный коммутатор: crossbar. Любой процессор системы получает данное по произвольному адресу памяти за одинаковое время, такая структура памяти называется: **UMA** - Uniform Memory Access (Architecture). Пример:HP-9000. Дальнейшее **масштабирование** (увеличение числа процессоров системы) SMP систем обеспечивается переходом к архитектуре памяти: **NUMA** - Non Uniform Memory Access. По схеме, называемой, этой иногда, кластеризацией SMP, соответствующие блоки памяти двух (или более) серверов соединяются кольцевой связью, обычно по GCI интерфейсу. При запросе данного, расположенного вне локального с сервере диапазона адресов, это данное по кольцевой связи переписывается дублируется в соответствующий блок локальной памяти, ту часть его, которая специально отводится для буферизации глобальных данных и из этого буфера поставляется потребителю. Эта буферизация прозрачна (невидима) пользователю, для которого вся память кластера имеет сквозную нумерацию, и время выборки данных, не локальных в сервере, будет равно времени выборки локальных данных при повторных обращениях к глобальному данному, когда оно уже переписано в буфер. Данный аппарат буферизации есть типичная схема кэш памяти. Так как к данным возможно обращение из любого процессора кластера, то буферизация, размножение данных требует обеспечение их когерентности. Когерентность данных состоит в том, что при изменении данного все его потребители должны получать это значение. Проблема когерентности усложняется дублированием данных еще и в процессорных кэшах системы. Системы, в которых обеспечена когерентность данных, буферизуемых в кэшах, называются кэш когерентными (cc-cache coherent), а архитектура памяти описываемого кластера: cc- NUMA (cache coherent Non Uniform Memory Access). Классической архитектурой принято считать систему SPP1000.
Другой источник:
Архитектура Система состоит из нескольких однородных процессоров и массива общей памяти (обычно из нескольких независимых блоков). Все процессоры имеют доступ к любой точке памяти с одинаковой скоростью. Процессоры подключены к памяти либо с помощью общей шины (базовые 2-4 процессорные SMP-сервера), либо с помощью crossbar-коммутатора (HP 9000). Аппаратно поддерживается когерентность кэшей.
Примеры HP 9000 V-class, N-class; SMP-сервера и рабочие станции на базе процессоров Intel (IBM, HP, Compaq, Dell, ALR, Unisys, DG, Fujitsu и др.).
Масштабируемость Наличие общей памяти сильно упрощает взаимодействие процессоров между собой, однако накладывает сильные ограничения на их число - не более 32 в реальных системах. Для построения масштабируемых систем на базе SMP используются кластерные или NUMA-архитектуры.
Операционная система Вся система работает под управлением единой ОС (обычно UNIX-подобной, но для Intel-платформ поддерживается Windows NT). ОС автоматически (в процессе работы) распределяет процессы/нити по процессорам (scheduling), но иногда возможна и явная привязка.
Модель программирования Программирование в модели общей памяти. (POSIX threads, OpenMP). Для SMP-систем существуют сравнительно эффективные средства автоматического распараллеливания.

29. Матричные мультипроцессорные системы.

Архитектура Система состоит из однородных вычислительных узлов (объединенных в матрицы или гиперкубы), включающих:

- один или несколько центральных процессоров (обычно RISC),
- **локальную память** (прямой доступ к памяти других узлов невозможен),
- коммуникационный процессор или сетевой адаптер
- иногда - жесткие диски (как в SP) и/или другие устройства В/В

К системе могут быть добавлены специальные узлы ввода-вывода и управляющие узлы. Узлы связаны через некоторую коммуникационную среду (высокоскоростная сеть, коммутатор и т.п.)
Примеры IBM RS/6000 SP2, Intel PARAGON/ASCI Red, CRAY T3E, Hitachi SR8000, транспьютерные системы Parsytec.
Масштабируемость Общее число процессоров в реальных системах достигает нескольких тысяч (ASCI Red, Blue Mountain).
Операционная система Существуют два основных варианта:

- Полноценная ОС работает только на управляющей машине (front-end), на каждом узле работает сильно урезанный вариант ОС, обеспечивающие только работу расположенной в нем ветви параллельного приложения. Пример: Cray T3E.
- На каждом узле работает полноценная UNIX-подобная ОС (вариант, близкий к кластерному подходу, однако скорость выше, чем в кластере). Пример: IBM RS/6000 SP + ОС AIX, устанавливаемая отдельно на каждом узле.

Модель программирования Программирование в рамках модели передачи сообщений (MPI, PVM, BSPlib)

30. Классификации вычислителей по Флинну.

Исходная схема: АЛУ + УУ <кд.....кд.....кд> ОЗУ
Данная классификация иллюстрируется схемами повышения производительности классического процессора путем увеличения количества функциональных устройств.
Увеличив число АЛУ, получим схему:
АЛУ + УУ <кд.....кд.....кд> ОЗУ
АЛУ <д.....д.....д> ОЗУ
По такой схеме создавалась система Эллиак 4 (отечественный аналог - ПС-2000), суперскалярные микропроцессоры.
Увеличив число УУ , то получим следующую схему:
АЛУ + УУ <кд.....кд.....кд> ОЗУ
УУ <к.....к.....к> ОЗУ
Некоторые исследователи отказывают данной схеме в праве на существование, другие, в качестве примера данной схемы приводят конвейерные АЛУ. (Например, сложение вещественных чисел можно реализовать последовательностью команд: выровнять мантиссы, сложить мантиссы, провести нормализацию результата.)
Наконец, можно просто умножать исходную схему:
АЛУ + УУ <кд.....кд.....кд> ОЗУ
АЛУ + УУ <кд.....кд.....кд> ОЗУ
По такой схеме реализуются все современные суперЭВМ
Кроме функционального различия, схемы отличаются характером нагрузки на ОЗУ- плотностью потоков команд и данных. По Флинну эта особенность и является главной чертой и она характеризует архитектуры ЭВМ по структуре используемых потоков (Stream) команд (Instruction) и данных (Data), каждый из которых может быть одиночным и множественным. Множественный поток определяется как возможность одновременной обработки сразу нескольких элементов соответствующего потока. Комбинация признаков дает четыре вида архитектур.
ОКОД одиночный поток команд, одиночный поток данных,
ОКМД одиночный поток команд, множественный поток данных,
МКОД множественный поток команд, одиночный поток данных,
МКМД множественный поток команд, множественный поток данных

31. Масштабируемость мультипроцессорных вычислителей.

См. *Вопросы 28-30.*

- Если память не общая, то масштабируемость очень высокая. Для MPP- *общее число процессоров в реальных системах достигает нескольких тысяч (ASCI Red, Blue Mountain).* Для кластеров и тем более гетерогенных вычислительных систем и тем более для GRID-систем масштабируемость очень высокая.
- Для систем с общем памятью все сложнее: если UMA, то масштабируемость низкая (не более 32 процессоров), если же сделать NUMA, то масштабируемость также становится высокой, однако ниже, чем в системах с локальной памятью, так как сложность организации когерентности (cc-NUMA) данных возрастает.

32. Управление памятью в мультипроцессорных системах.

UMA, NUMA, cc-NUMA. Подробнее см. *Вопрос 28.*

33. Когерентность данных.

Когерентность данных состоит в том, что при изменении данного все его потребители должны получать это значение. Проблема когерентности усложняется дублированием данных еще и в процессорных кэшах системы. Системы, в которых обеспечена когерентность данных, буферизуемых в кэшах, называются кэш когерентными (cc-cache coherent), а архитектура памяти описываемого кластера: cc- NUMA (cache coherent Non Uniform Memory Access). Классической архитектурой принято считать систему SPP1000. Подробнее см. *Вопрос 28*
Другой источник:
Для обеспечения целостности данных, имеющих несколько копий в разных процессорах и в памятях разных уровней иерархии, разрабатываются спецификации протоколов когерентности.
Существуют несколько классов таких протоколов:

1. Протоколы на основе справочника (directory based). Информация о состоянии блока физической памяти содержится только в одном месте, называемом справочником (физически справочник может быть распределён по узлам системы).
2. Протоколы наблюдения (snooping). Каждый кэш, который содержит копию данных некоторой части физической памяти, имеет также соответствующую копию служебной информации (биты состояния). Контроллеры кэш-памятей обмениваются между собой служебной информацией о состоянии блока данных.
Задачей протокола когерентности кэш-памяти является координация доступа к разделяемым блокам памяти. Кэш-памяти процессоров обмениваются информацией для того, чтобы определить какой процессор в настоящий момент является собственником данных. Одним из наиболее полных протоколов является протокол MOESI (Modified, Owned, Exclusive, Shared, Invalid), в котором доступ процессора к блоку данных определяется состоянием этого блока и типом запроса. Работа данного протокола описывается поведением конечного автомата с 5 состояниями. На верхнем уровне абстракции проверка правильности работы сводится к моделированию поведения автомата и контролю корректности его состояний и не представляет существенной сложности. Однако, детализация работы протокола в реальном устройстве приводит к значительным усложнениям полученной картины. Так, даже для простейшего протокола типа MSI (однопроцессорная система) имеем 11 состояний (8 из которых переходных), 13 возможных событий и 21 действие.

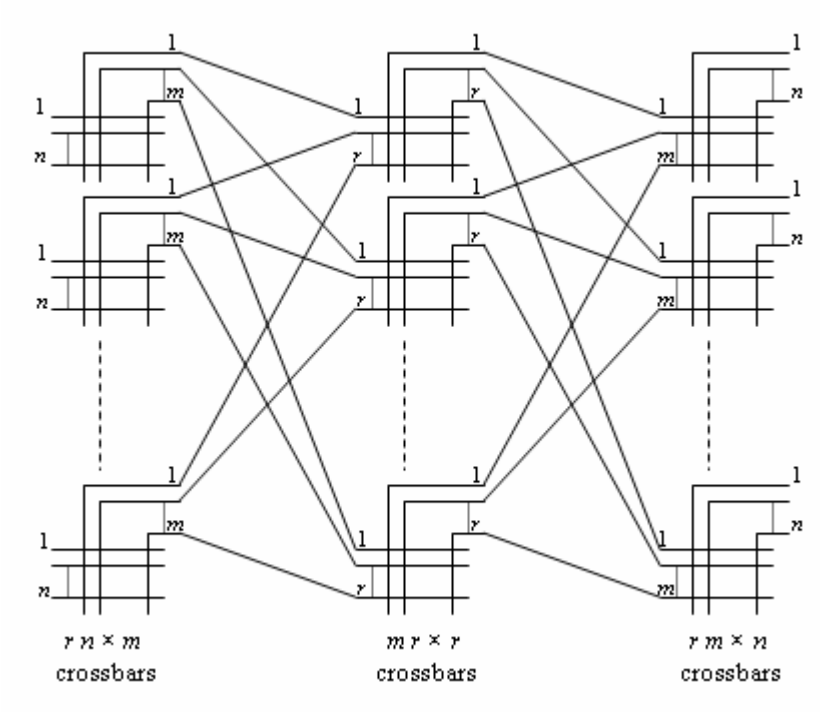
34. Типы внутренних связей.

Сети обмена делятся на три типа: **шины, статические сети, динамические сети.**
Быстродействие сети обмена определяется быстродействием узлов и связей между ними, параметры последних определяются: пропускной способностью непрерывного потока данных и максимальным количеством самых маленьких пакетов, которые можно передать за единицу времени (время передачи пакета нулевой длины, задержка). Некоторые сетевые интерфейсы:

Технология	Mbyte/s	Задержка (мкс)
Fast Ethernet	12.5	156
Gigabit Ethernet	125	33
Myrinet	245	6
SCI	400	1.5

Шины
Система связи ПЭ через коммутирующую общую шину отличаются от других схем простотой, однако ее производительность ограничивается необходимостью обеспечивать в любой момент времени не более одного запроса на передачу информации. Общая шина с центральным коммутатором обеспечивает арбитраж запросов процессоров на передачу информации - выделение одного из запросов на доступ к шине и обеспечение монопольного права на владения шиной на все время требуемого обмена. Шины - пропускная способность 100 Мбайт/с для 2 -20 ПЭ (Multimax). Число процессоров в таких системах всегда ограничено.
Статические сети
Статические сети имеют жестко фиксированные соединения, вход и выход зафиксированы без возможности переключения. Наиболее простой топологией сети является линейка – одномерная сетка. Для обеспечения передачи информации между несмежными узлам используются транзитные пересылки. Для цепочки из M узлов наиболее медленная пересылка есть пересылка между конечными ПЭ линейки и она потребует (M-1) транзитных пересылок. Число таких пересылок для любой статической топологии сети считается ее параметром и называется – диаметром сети. Если связать конечные ПЭ линейки, то такая топология - кольцо будет иметь меньший диаметр. Сети могут иметь вид: одномерный линейный массив, двумерное кольцо, звезда, сетка и гексагональный массив, дерево, трехмерное полностью связанное хордовое кольцо, 3 -мерный куб, сети из циклически связанных 3-мерных кубов, D - мерный массив с топологией гиперкуба, тасовка (совершенная, обменная). Недостаток - необходимость маршрутизации транзитных сообщений. По топологии гиперкуба, каждое ПЭ связывается со своим ближайшем соседом в n мерном пространстве. У каждого узла в двумерном гиперкубе имеются два ближайших соседа, в трехмерном - три, в четырехмерном - четыре.
Динамические сети
Динамические сети - сети с возможностью динамического (коммутируемого) соединения узлов сети друг с другом. Особое место занимает полный коммутатор.
Полный коммутатор обеспечивает полную связность каждого узла в сети, причем, обеспечивает независимость (не блокируемость) пересылок. Недостаток: высокая стоимость аппаратуры и ограниченная размерность.
Перечисленные выше однокаскадные сети обмена содержат фиксированное число каскадов или один каскад переключателей. Многокаскадные сети могут быть получены комбинацией некоторых однокаскадных сетей и могут составить конкуренцию полному коммутатору. Например, стандартная сеть Клоша (Клоса) может иметь нечетное число каскадов и строится из сетей меньших размеров. Clos networks have three stages: the ingress stage, middle stage, and the egress stage. Each stage is made up of a number of crossbar switches (see diagram below), often just called crossbars. Each call entering an ingress crossbar switch can be routed through any of the available middle stage crossbar switches, to the relevant egress crossbar switch. A middle stage crossbar is available for a particular new call if both the link connecting the ingress switch to the middle stage switch, and the link connecting the middle stage switch to the egress switch, are free. Clos networks are defined by three integers n, m, and r. n represents the number of sources which feed into each of r ingress stage crossbar switches. Each ingress stage crossbar switch has m outlets, and there are m

centre stage crossbar switches. There is exactly one connection between each ingress stage switch and each middle stage switch. There are r egress stage switches, each with m inputs and n outputs. Each middle stage switch is connected exactly once to each egress stage switch.



35. Статические и динамические коммуникаторы.

См Вопрос 34.

36. Параметры статических коммутационных сетей.

См Вопрос 34.

37. Топологии линейки, решетки, пирамиды.

Наиболее простой топологией сети является **линейка** – одномерная сетка. Для обеспечения передачи информации между несмежными узлам используются транзитные пересылки. Для цепочки из M узлов наиболее медленная пересылка есть пересылка между конечными ПЭ линейки и она потребует (M-1) транзитных пересылок. **Решетка** –таблица, в которой узлы связаны только со своими соседями вверх и вниз – то есть матрица. Удобно для решения и представления задач с изображением и другими пространственными задачами, где решение тесно связано с решением соседних по пространству частей. **Деревья** имеют хорошую структуру для большого числа задач, в которых информация сортируется, сравнивается или каким-либо образом уплотняется и реорганизуется, а также где она запоминается, извлекается или передается. Матрицы имеют хорошую структуру для локальной передачи информации.

Наилучшими считаются **пирамидальные** многопроцессорные системы, поскольку они очень эффективны не только при параллельной локальной обработке,но и при глобальных передачах и преобразованиях информации. Древовидная топология **пирамиды** определяет возможность накопления и объединения информации по мере поэтапного преобразования данных. Также наиболее актуально для изображения. Матричные, конвейерные и в особенности пирамидальные структуры обеспечивают увеличение производительности и вычислительной мощности на несколько порядков по сравнению с традиционными ЭВМ с одним ЦП. Они особенно пригодны для обработки изображений, распознавания образов и в системах технического зрения. Они также хорошо соответствуют требованиям технологии СБИС благодаря своей регулярной микромодульной структуре.

38. Топология гиперкуба.

Двоичный гиперкуб - распространенная, хорошо масштабируемая и очень удачная топология (рис. 25). В ней 2k процессоров организованы в k-мерный гиперкуб. У каждого процессора k непосредственных соединений (k = log2N), максимальное расстояние между узлами k. Для адресации узлов в гиперкубе каждому узлу присваивается идентификационный номер (ID). Двоичное представление ID соседних узлов отличается одним разрядом. Алгоритм пересылки сообщения от одного узла к другому основан на побитовом сравнении двоичных представлений ID текущего узла и адресата. Для каждого процессора легко определить всех его соседей - они отличаются лишь значением какого-либо одного разряда в двоичном представлении ID. Каждая грань k-мерного гиперкуба является гиперкубом размерности k-1. *См. также Таненбаум, стр 43.*

39. Согласование сеточных топологий со структурой гиперкуба.

Сеточная топология (mesh), при которой компьютеры связываются между собой не одной, а многими линиями связи, образующими сетку. В полной сеточной топологии каждый компьютер напрямую связан со всеми остальными компьютерами. В этом случае при увеличении числа компьютеров резко возрастает количество линий связи. Кроме того, любое изменение в конфигурации сети требует внесения изменений в сетевую аппаратуру всех компьютеров, поэтому полная сеточная топология не получила широкого распространения. Частичная сеточная топология предполагает прямые связи только для самых активных компьютеров, передающих максимальные объемы информации. Остальные компьютеры соединяются через промежуточные узлы. Сеточная топология позволяет выбирать маршрут для доставки информации от абонента к абоненту, обходя неисправные участки. С одной стороны, это увеличивает надежность сети, с другой же – требует существенного усложнения сетевой аппаратуры, которая должна выбирать маршрут.

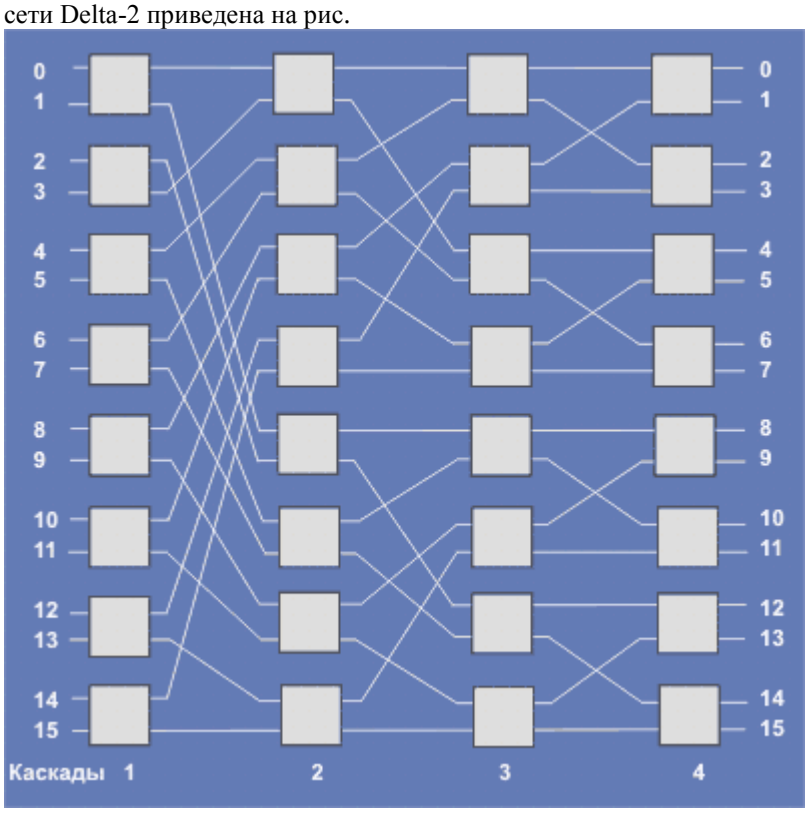
40. Перекрестный коммутатор.

Crossbar switches have a characteristic matrix of switches between the inputs to the switch and the output of the switch. If the switch has M inputs and N outputs, then a crossbar has a matrix with M x N cross-points or places where there the "bars" "cross". A given crossbar is a single layer, non-blocking switch. Collections of crossbars can be used to implement multiple layer and/or blocking switches. Дополнительно см. *Вопрос 34.*

41. Многокаскадные коммутационные сети.

Сначала см. *Вопрос 34.*

Источник 1: Среди традиционных сетей особое положение занимают сети, базирующиеся на большом числе идентичных ключевых элементов. Это сети с многокаскадными связями (MIN – Multistage Interconnection Network). Идеология таких сетей используется при построении коммутаторов ATM. Из таких сетей наиболее известной является Delta Banyan (Batcher-switch). Эта сеть содержит регулярную структуру N*N переключателей с двух направлений на два. Сеть содержит log2N число каскадов, каждый из которых имеет N/2 переключателей. Для управления маршрутизацией сообщений в этой сети необходимо log2N бит информации. Схема 4-каскадной сети Delta-2 приведена на рис.



Источник 2: Многокаскадные сети основаны на использовании 2x2 перекрестных переключателей - коммутаторов. В таких сетях на одном конце находятся процессоры, на другом - процессоры или другие узлы, а между ними располагаются переключатели. При передаче информации от узла к узлу переключатели устанавливаются таким образом, чтобы обеспечить требуемое соединение. На приведена схема соединения "омега", являющейся сетью с блокировкой, в которой один маршрут передачи данных может заблокировать другие маршруты. Для соединения n процессоров с n модулями памяти требуется (n/2)*log2n переключателей.

Матричный коммутатор обеспечивает полную связность, т.е. любой процессорный элемент (ПЭ) может связаться с любым другим процессорным элементом или модулем памяти (МП). Естественно, в ситуации, когда два процессора захотят работать с одним модулем памяти, один из них будет заблокирован. На пересечении линий располагаются элементарные точечные переключатели, которые разрешают или запрещают передачу информации.

Недостаток - большой объем необходимого оборудования и, как следствие, ограниченная масштабируемость системы. Для связи n процессоров с n модулями памяти необходимо n*2 элементарных переключателей.

42. Пиковая производительность.

Оценка производительности вычислительных систем имеет два аспекта: оценка пиковой производительности – номинального быстродействия системы и получение оценок максимальной - "реальной" производительности. Если номинальная(**пиковая, предельная**) оценка однозначно определяется техническими параметрами оборудования, то вторая характеристика указывает производительность системы в реальной среде применения. Для оценки производительности вычислительных систем в TOP500 используются обозначения Rpeak – пиковая, предельная производительность и Rmax – максимальная производительность при решении задачи Linpack.

Наиболее абстрактной единицей измерения производительности микропроцессоров является **тактовая частота** ПК, частота тактового генератора (*clock rate,).* Любая операция в процессоре не может быть выполнена быстрее, чем за один такт (период) генератора. Итак, минимальное время исполнения одной логической операции (переключение транзистора) - один такт. Тактовая частота измеряется в Герцах – число тактов в секунду.

Другой обобщенной мерой производительности ПК может служить **число команд, выполняемые в единицу времени.** Для вычислителей фон-Неймановской архитектуры скорость выполнения команд уже может быть параметром, который может быть использован для оценки времени выполнения программы. Этот параметр - **MIPS** (Million Instruction Per Second)- миллион операций (команд, инструкций ЭВМ)/сек. Так как время выполнения различных команд может различаться, то данных параметр сопровождался разного вида уточнениями (логических команд, заданной смеси команд и т.д.), а также наиболее известной здесь мерой в 1 MIPS – это производительность вычислителя VAX 11/780. Итак, данный параметр также достаточно условен.

Так как для большинства вычислительных алгоритмов существуют оценки числа арифметических операций, необходимых для выполнения расчетов, данная мера и может служить тем показателем, который и интересует пользователей в первую очередь. Это – **MFLPOPS** (Million of Floating point Operation Per Second – **Мегафлопс**)- миллион операций на данных с плавающей запятой/сек, единица быстродействия ЭВМ в операциях с плавающей запятой, есть также единицы - GFLPOPS и TFLPOPS (терафлопс = 10**12 оп./сек.).

43. Методы оценки производительности.

Обычно, рассматриваются три подхода к оценке производительности:

- на базе аналитических модели (системами массового обслуживания);
- имитационное моделирование;
- измерения.

Первый подход обеспечивает наиболее общие и наименее точные результаты, последние, наоборот, - наименее общие и наиболее точные. Измерения проводятся контрольными (тестовыми) программами.

Бенч-марк (Benchmark) - эталон:

- стандарт, по которому могут быть сделаны измерения или сравнения;
- процедура, задача или тест, которые могут быть использованы для сравнения систем или компонентов друг с другом или со стандартом как в п.1.

Для повышения общности и представительности оценки производительности контрольные программы можно разделить на:

- программы нижнего уровня. Эти программы тестируют основные машинные операции - +,/,* , с учетом времени доступа к памяти, работу кэша, характеристики ввода/вывода.
- ядра программ. Ядра программ - короткие характерные участки программ, например, Ливерморские фортрановские ядра (24 ядра) , Эймсовские ядра НАСА, синтетический тест Ветстоун (Whetstone).
- основные подпрограммы и типовые процедуры; Примером основных подпрограмм могут быть программы Линпак (Linpack) , программы типа быстрого преобразования Фурье. Программа Линпак - процедура решения системы линейных уравнений методом исключения Гаусса. В этой схеме вычислений точно известно число операций с плавающей точкой, которое зависит от размерности массивов – параметров. Стандартные значения размерностей 100 или 1000. Для параллельных ЭВМ имеется соответствующая версия теста.
- полные основные прикладные программы; В качестве примеров программ этого уровня приводятся Лос-Аламосские тестовые программы моделирования поведения плазмы и программы гидродинамики.
- перспективные прикладные программы.

44. Закон Амдала.

Закон Амдала показывает коэффициент ускорения выполнения программы на параллельных системах в зависимости от степени распараллеливания программы. Пусть: N - число процессоров системы, P - доля распараллеливаемой части программы, а S = 1-P - доля скалярных операций программы, выполняемых без совмещения по времени.(S+P = 1 , S,P >= 0). Тогда, по Амдалу, общее время выполнения программы на однопроцессорном вычислителе S+P, на мультипроцессоре: S+P/N, а ускорение при этом есть функция от P: Sp = (S+P)/(S+P/N) = 1/(S+P/N).

Из формулы закона Амдала следует,что при:

P = 0 S = 1 - ускорения вычисления нет, а
P = 1 S = 0 - ускорение вычислений в N раз
Если P = S = 0.5, то даже при бесконечном числе процессоров ускорение не может быть более чем в 2 раза.

45. Принципы потоковой обработки информации.

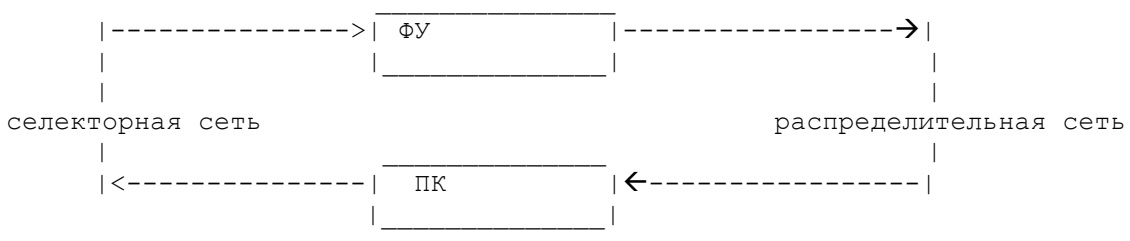
Потоковая архитектура (data-flow) вычислительных систем обеспечивает интерпретацию алгоритмов на графах, управляемых данными. Идеи потоковой обработки информации, организации вычислений, управляемых потоками данных можно рассмотреть на примере организации ввода и суммирования трех чисел. Традиционная схема вычислений может быть представлена так: ввод (a); ввод (в); ввод (c); s = a+b; s = s+c. Если ввод данных может производиться асинхронно, то организовать параллельное программирования данного алгоритма не просто. Параллельный алгоритм может быть записан в виде потока данных на графе:

```
ввод (а)     ввод (в)     ввод (с)
  а+в          а+с          в+с
(в+с)+а      (а+с)+в      (а+в)+с
```

Здесь, начальные вершины - ввод, затем каждое введенное данное размножается на три и они передаются на вершины, обеспечивающие суммирование. Теперь, в любом порядке поступления данных отсутствуют задержки вычислений для получения результата. Data-flow программы записываются в терминах графов. В вершинах графа находятся команды, состоящие, например, из оператора, двух операндов (для двуместных операций), возможно, литеральных, или шаблонов для заранее неизвестных данных и ссылки, определяющей команду - наследника и позицию аргумента в ней для результата.

46. Схемы потоковых вычислителей.

Основными компонентами потоковой ВС являются:
- память команд (ПК),
- селекторная (арбитражная) сеть,
- множество исполнительных (функциональных) устройств (ФУ),
- распределительная сеть.



Память команд состоит из "ячеек" активной памяти, каждая из которых может содержать одну команду вида <метка>: <операция>,<операнд1>,..,<операндК>,<адр_рез1>,..,<адр. _рез.М>, где адреса результатов являются адресами ячеек памяти. С каждой командой связан подсчитывающий элемент, непрерывно ожидающий прибытие аргументов, который пересылает команду на выполнение при наличии полного комплекта аргументов. Активный характер памяти заключается в том, что ячейка, обладающая полным набором операндов, переходит в возбужденное состояние и передает в селекторную сеть информационный пакет, содержащий необходимую числовую и связующую информацию о команде.
Селекторная сеть обеспечивает маршрут от каждой командной ячейки к выбранному, в соответствии с кодом операции, исполнительному (функциональному) устройству из множества устройств. Пакет поступает на одно из исполнительных устройств, где соответствующая операция выполняется и результат подается в распределительную сеть.
Распределительная сеть обрабатывает результирующий пакет, состоящий из результатов вычислений и адресов назначения. В зависимости от содержимого пакета, результат вычислений поступает в соответствующие ячейки памяти команд, создавая, тем самым, условия возможности их активизации.
Потоковая архитектура (data-flow), как одна из альтернатив фон-Неймановской, обладает следующими характерными чертами:
- отсутствие памяти как пассивного устройства, хранящего потребляемую информацию,
- отсутствие счетчика команд (и, следовательно, последовательной обработки команд программы, разветвлений по условию и т.д.).
Потоковые вычислительные системы позволяют использовать параллелизм вычислительных алгоритмов различных уровней, потенциально достигать производительность, недоступную традиционным вычислительным системам. Основные проблемы, препятствующие развитию потоковых машин:

1. Не решена проблема создания активной памяти большого объема, допускающей одновременную активизацию большого количества операций.
2. Создание широкополосных распределительных и селекторных сетей потоковых машин и систем управления коммуникационной сетью является сложной задачей.
3. Обработка векторных регулярных структур через механизмы потока данных менее эффективна, чем традиционные решения.
4. Языки программирования для потоковых машин существуют, в основном, в виде графических языков машинного уровня. Языки типа SISAL, ориентируемые на описания потоковых алгоритмов, достаточно сложны для программистов.

47. Нейронные сети.

Одно из наиболее перспективных направлений разработки принципиально новых архитектур вычислительных систем тесно связано с созданием компьютеров нового поколения на основе принципов обработки информации, заложенных в искусственных нейронных сетях (НС). Первые практические работы по искусственным нейросетям и нейрокомпьютерам начались еще в 40-50-е годы. Под нейронной сетью обычно понимают совокупность элементарных преобразователей информации, называемых «нейронами», которые определенным образом соединены друг с другом каналами обмена информации – «синаптическими связями». Нейрон, по сути, представляет собой элементарный процессор, характеризующийся входным и выходным состоянием, передаточной функцией (функция активации) и локальной памятью. Состояния нейронов изменяются в процессе функционирования и составляют кратковременную память нейросети. Каждый нейрон вычисляет взвешенную сумму пришедших к нему по синапсам сигналов и производит над ней нелинейное преобразование. При пересылке по синапсам сигналы умножаются на некоторый весовой коэффициент. В распределении весовых коэффициентов заключается информация, хранящаяся в ассоциативной памяти НС. Основным элементом проектирования сети является ее обучение. При обучении и переобучении НС ее весовые коэффициенты изменяются. Однако они остаются постоянными при функционировании нейросети, формируя долговременную память. НС может состоять из одного слоя, из двух, из трех и большего числа слоев, однако, как правило, для решения практических задач более трех слоев в НС не требуется. Число входов НС определяет размерность гиперпространства, в котором входные сигналы могут быть представлены точками или гиперобластями из близко расположенных точек. Количество нейронов в слое сети определяет число гиперплоскостей в гиперпространстве. Вычисление взвешенных сумм и выполнение нелинейного преобразования позволяют определить, с какой стороны от той или иной гиперплоскости находится точка входного сигнала в гиперпространстве. Искусственные нейронные сети отличаются удивительными свойствами. Они не требуют детализированной разработки программного обеспечения и открывают возможности решения задач, для которых отсутствуют теоретические модели или эвристические правила, определяющие алгоритм решения. Такие сети обладают способностью адаптироваться к изменениям условий функционирования, в том числе к возникновению заранее непредусмотренных факторов. По своей природе НС являются системами с очень высоким уровнем параллелизма.
В нейрокомпьютерах используются принципы обработки информации, осуществляемые в реальных нейронных сетях. Эти принципиально новые вычислительные средства с нетрадиционной архитектурой позволяют выполнять высокопроизводительную обработку информационных массивов большой размерности. В отличие от традиционных вычислительных систем, нейросетевые вычислители, аналогично нейронным сетям, дают возможность с большей скоростью обрабатывать информационные потоки дискретных и непрерывных сигналов, содержат простые вычислительные элементы и с высокой степенью надежности позволяют решать информационные задачи обработки данных, обеспечивая при этом режим самоперестройки вычислительной среды в зависимости от полученных решений.
Вообще говоря, под термином «нейрокомпьютер» в настоящее время подразумевается довольно широкий класс вычислителей. Это происходит по той простой причине, что формально нейрокомпьютером можно считать любую аппаратную реализацию нейросетевого алгоритма, от простой модели биологического нейрона до системы распознавания символов или движущихся целей. Нейрокомпьютеры не являются компьютерами в общепринятом смысле этого слова. В настоящее время технология еще не достигла того уровня развития, при котором можно было бы говорить о нейрокомпьютере общего назначения (который являлся бы одновременно искусственным интеллектом). Системы с фиксированными значениями весовых коэффициентов – вообще самые узкоспециализированные из нейросетевого семейства. Обучающиеся сети более адаптированы к разнообразию решаемых задач. Обучающиеся сети более гибки и способны к решению разнообразных задач. Таким образом, построение нейрокомпьютера – это каждый раз широчайшее поле для исследовательской деятельности в области аппаратной реализации практически всех элементов НС.
В начале 21 века, в отличие от 40-50-х годов прошлого столетия, существует объективная практическая потребность научиться создавать нейрокомпьютеры, т.е. необходимо аппаратно реализовать довольно много параллельно действующих нейронов, с миллионами фиксированных или параллельно адаптивно модифицируемых связей-синапсов, с несколькими полносвязными слоями нейронов.
В то же время физические возможности технологии интегральной электроники не безграничны. Геометрические размеры транзисторов больше нельзя физически уменьшать: при технологически достижимых размерах порядка 1 мкм и меньше проявляются физические явления, незаметные при больших размерах активных элементов – начинают сильно сказываться квантовые размерные эффекты. Транзисторы перестают работать как транзисторы.

Для аппаратной реализации НС необходим новый носитель информации. Таким новым носителем информации может быть свет, который позволит резко, на несколько порядков, повысить производительность вычислений. Единственной технологией аппаратной реализации НС, способной в будущем прийти на смену оптике и оптоэлектронике, является нанотехнология, способная обеспечить не только физически предельно возможную степень интеграции субмолекулярных квантовых элементов с физически предельно возможным быстродействием, но и столь необходимую для аппаратной реализации НС трехмерную архитектуру. См. также *Вопрос 48.*

48. Области применения нейронных сетей.

Возьмем классическую задачу распознавания образов: определение принадлежности точки одному из двух классов. Такая задача естественным образом решается с помощью одного нейрона. Он позволит разделить гиперпространство на две непересекающиеся и невложенные гиперобласти. Входные сигналы в задачах, решаемых с помощью нейросетей, образуют в гиперпространстве сильно вложенные или пересекающиеся области, разделить которые с помощью одного нейрона невозможно. Это можно сделать, только проведя нелинейную гиперповерхность между областями. Ее можно описать с помощью полинома n-го порядка. Однако степенная функция слишком медленно считается и поэтому очень неудобна для вычислительной техники. Альтернативным вариантом является аппроксимация гиперповерхности линейными гиперплоскостями. Понятно, что при этом точность аппроксимации зависит от числа используемых гиперплоскостей, которое, в свою очередь, зависит от числа нейронов в сети. Отсюда возникает потребность в аппаратной реализации как можно большего числа нейронов в сети. Количество нейронов в одном слое сети определяет ее разрешающую способность. Однослойная НС не может разделить линейно зависимые образы. Поэтому важно уметь аппаратно реализовывать многослойные НС.

Длительное время считалось, что нейрокомпьютеры эффективны для решения так называемых неформализуемых и плохо формализуемых задач, связанных с необходимостью включения в алгоритм решения задачи процесса обучения на реальном экспериментальном материале. В первую очередь к таким задачам относилась задача аппроксимации частного вида функций, принимающих дискретное множество значений, т. е. задача распознавания образов.

В настоящее время к этому классу задач добавляется класс задач, иногда не требующий обучения на экспериментальном материале, но хорошо представимый в нейросетевом логическом базисе. К ним относятся задачи с ярко выраженным естественным параллелизмом обработки сигналов, обработка изображений и др. Подтверждением точки зрения, что в будущем нейрокомпьютеры будут более эффективными, чем прочие архитектуры, может, в частности, служить резкое расширение в последние годы класса общематематических задач, решаемых в нейросетевом логическом базисе. К ним, кроме перечисленных выше, можно отнести задачи решения линейных и нелинейных алгебраических уравнений и неравенств большой размерности; систем нелинейных дифференциальных уравнений; уравнений в частных производных; задач оптимизации и других задач.

49. Модели программирования для систем с разделяемой, распределенной памятью.

Программирование в модели общей памяти. (POSIX threads, OpenMP). Для SMP-систем существуют сравнительно эффективные средства автоматического распараллеливания. Подробнее см. *Вопросы* .

Программирование в модели распределенной памяти. Программирование в рамках модели передачи сообщений (MPI, PVM, BSPlib).

Эта модель схожа с парадигмой последовательного программирования. Программист имел упрощенную точку зрения на целевую машину, которая имеет единственный процессор и адресуется к определенному объему памяти. Поэтому он писал единственную программу, чтобы выполнять ее на этом процессоре. В сущности, парадигма может быть обеспечена разными путями, возможно, в системе разделения времени, где другие процессы разделяют процессор и память, но программисту нет необходимости вдаваться такие детали реализации, если он уверен, что программа или разработанный алгоритм в принципе могут быть перенесены на любую логическую архитектуру -- это сущность парадигмы.

Парадигма передачи сообщений является развитием этой идеи применительно к целям параллельного программирования. Некоторые положения последовательной парадигмы сохраняются. Так, программист представляет несколько процессоров, каждый со своим адресным пространством, и составляет программу, чтобы запустить ее на каждом процессоре. Параллельное программирование по определению требует взаимодействия между процессорами для решения задачи, используя некоторую связь. Основное положение парадигмы передачи сообщений - процессы общаются путем передачи друг другу сообщений. Модель передачи сообщений не вводит понятие разделяемого адресного пространства или непосредственного доступа одного процессора к памяти другого - все не относящееся к передаче сообщений, находится вне парадигмы. Поскольку программы, выполняющиеся на отдельных процессорах, связаны, операции передачи сообщений представляют собой вызовы подпрограмм.

Для тех, кто имеет опыт работы с сетями, системами клиент-сервер или объектно-ориентированными программами, парадигма передачи сообщений не преподнесет ничего нового. С недавнего времени парадигма программирования с передачей сообщений стала популярной. Одной из причин этого стало увеличение числа платформ, поддерживающих модель передачи сообщений. Программы, написанные с использованием передачи сообщений, могут выполняться на распределенных многопроцессорных комплексах или системах с разделяемой

памятью, сетях и даже на однопроцессорых системах. Применяя парадигму, программист знает, что его алгоритмы должны быть в принципе переносимы на любую платформу, поддерживающую модель передачи сообщений. Модель передачи сообщений популярна не потому, что достаточно проста, но потому, что она фундаментальна.
Программирование в модели обработки данных – параллелизм по данным. Подробнее см. *Вопросы 45, 46.*

50. Разделение последовательных программ на параллельные нити.

В системах с общей памятью
В принципе, для распараллеливания программ можно использовать механизм порождения процессов. Однако этот механизм не очень удобен, поскольку каждый процесс функционирует в своем адресном пространстве, и основное достоинство этих систем - общая память - не может быть использован простым и естественным образом. Для распараллеливания программ используется механизм порождения нитей (threads) - легковесных процессов, для которых не создается отдельного адресного пространства, но которые на многопроцессорных системах также распределяются по процессорам. В языке программирования С возможно прямое использование этого механизма для распараллеливания программ посредством вызова соответствующих системных функций, а в компиляторах с языка FORTRAN этот механизм используется либо для автоматического распараллеливания, либо в режиме задания распараллеливающих директив компилятору (такой подход поддерживают и компиляторы с языка С).
Все производители симметричных мультипроцессорных систем в той или иной мере поддерживают стандарт POSIX Pthread и включают в программное обеспечение распараллеливающие компиляторы для популярных языков программирования или предоставляют набор директив компилятору для распараллеливания программ. В частности, многие поставщики компьютеров SMP архитектуры (Sun, HP, SGI) в своих компиляторах предоставляют специальные директивы для распараллеливания циклов. Однако эти наборы директив, во-первых, весьма ограничены и, во-вторых, несовместимы между собой. В результате этого разработчикам приходится распараллеливать прикладные программы отдельно для каждой платформы.

51. Ограничения на распараллеливание циклов.

Распараллеливание циклов возможно, если все итерации цикла независимы. Тело цикла не должны содержать:

- операторов перехода
- операторов ввода-вывода

Индексные выражения не должны иметь индекс в индексе A(B(C))

+ см. вопрос 64

52. Синхронизация параллельных процессов. Барьеры.

Барьеры – весьма своеобразное средство синхронизации. Идея его в том, чтобы в определенной точке ожидания собралось заданное число потоков управления. Только после этого они смогут продолжить выполнение. (Поговорка "семеро одного не ждут" к барьерам не применима.)
Барьеры полезны для организации коллективных распределенных вычислений в многопроцессорной конфигурации, когда каждый участник (поток управления) выполняет часть работы, а в точке сбора частичные результаты объединяются в общий итог.
Функции, ассоциированные с барьерами, подразделяются на следующие группы.

инициализация и разрушение барьеров: pthread_barrier_init(), pthread_barrier_destroy()

```
#include <pthread.h>

int pthread_barrier_init (
pthread_barrier_t *restrict barrier,
const pthread_barrierattr_t
*restrict attr,
unsigned count);

int pthread_barrier_destroy (
pthread_barrier_t *barrier);
```

синхронизация на барьере: pthread_barrier_wait() (см. листинг 2.37);

```
#include <pthread.h>
int pthread_barrier_wait (
pthread_barrier_t *barrier);
```

инициализация и разрушение атрибутных объектов барьеров: pthread_barrierattr_init(), pthread_barrierattr_destroy() (см. листинг 2.38);

```
#include <pthread.h>

int pthread_barrierattr_init (
pthread_barrierattr_t *attr);

int pthread_barrierattr_destroy (
pthread_barrierattr_t *attr);
```

опрос и установка атрибутов барьеров в атрибутных объектах: pthread_barrierattr_getpshared(), pthread_barrierattr_setpshared() (см. листинг 2.39).

```
#include <pthread.h>

int pthread_barrierattr_getpshared
(const pthread_barrierattr_t
      *restrict attr,
int *restrict pshared);

int pthread_barrierattr_setpshared
(pthread_barrierattr_t *attr,
      int pshared);
```

Обратим внимание на аргумент count в функции инициализации барьера pthread_barrier_init(). Он задает количество синхронизируемых потоков управления. Столько потоков должны вызвать функцию pthread_barrier_wait(), прежде чем каждый из них сможет успешно завершить вызов и продолжить выполнение. (Разумеется, значение count должно быть положительным.)

Когда к функции pthread_barrier_wait() обратилось требуемое число потоков управления, одному из них (стандарт POSIX-2001 не специфицирует, какому именно) в качестве результата возвращается именованная константа PTHREAD_BARRIER_SERIAL_THREAD, а всем другим выдаются нули. После этого барьер возвращается в начальное (инициализированное) состояние, а выделенный поток может выполнить соответствующие объединительные действия.

```
if ((status = pthread_barrier_wait(
  &barrier)) ==
        PTHREAD_BARRIER_SERIAL_THREAD) {
 /* Выделенные (обычно – объединительные) */
 /* действия.     */
 /* Выполняются каким-то одним потоком */
 /* управления */

} else {
    /* Эта часть выполняется всеми */
    /* прочими потоками */
    /* управления */
    if (status != 0) {
    /* Обработка ошибочной ситуации */
    } else {
        /* Нормальное "невыделенное" */
        /* завершение ожидания */
        /* на барьере */
    }
}

/* Повторная синхронизация – */
/* ожидание завершения выделенных действий */
status = pthread_barrier_wait (&barrier);
/* Продолжение параллельной работы */
    . . .
```

Отметим, что для барьеров отсутствует вариант синхронизации с контролем времени ожидания. Это вполне понятно, поскольку в случае срабатывания контроля барьер окажется в неработоспособном состоянии (требуемое

число потоков, скорее всего, уже не соберется). По той же причине функция pthread_barrier_wait() не является точкой терминирования – "оставшиеся в живых" не переживут потери товарища...

53. Критические секции. Двоичные и общие семафоры.

Процессы для своей работы могут использовать логические и физические ресурсы вычислительной системы и ее окружения, причем ресурсы могут быть: поделены между процессами и закреплены постоянно (на все время работы процесса) или использованы всеми или некоторыми процессами по-очереди. Некоторые ресурсы могут быть общими и допускать параллельное обслуживание процессов. Ресурс, который допускает обслуживание только одного процесса в текущее время, называется критическим ресурсом.

Участки программы процесса, в которых происходит обращение к критическим ресурсам, называются критическими участками. Такие участки должны быть выполнены в режиме "взаимного исключения", т.е. в каждый момент времени не более чем один процесс может быть занят выполнением своего, критического относительно некоторого ресурса, участка. Проблема критического участка - программирование работы критических участков в режиме взаимного исключения решается с помощью семафоров. Общий семафор - это целочисленная переменная, над которой разрешены только две неделимые операции P и V. Аргументом у этих операций является семафор. Определять семантику этих операций можно только для семафоров - положительных чисел, или включать и отрицательный диапазон. Операции могут быть определены так:

P(S) - декремент и анализ семафора

```
S := S-1
IF (S < 0) THEN  <блокировка текущего процесса>
ENDIF
```

V(S) - инкремент семафора

```
S := S+1
IF S <= 0 THEN <запустить блокированный  процесс>
ENDIF
```

P и V операции неделимы: в каждый момент только один процесс может выполнять P или V операцию над данным семафором.

Если семафор используется как счетчик ресурсов, то:

S >= 1 - есть некоторое количество (S) доступных ресурсов,

S = 0 - нет доступного ресурса,

S < 0 - один или несколько (S) процессов находятся в очереди к ресурсу.

Вводится, также, операция инициализации семафора (присвоение начального значения).

Общий семафор - избыточный, т.к. его можно реализовать через двоичные семафоры, которые принимают только два значения 0,1.

Семафоры позволяют:

- синхронизировать работу процессов,
- управлять распределением ресурсом (использованием положительного диапазона значений семафора как счетчика ресурсов) ,
- организовывать очередь блокированных процессов (отрицательные значения семафора показывают число блокированных процессов).

В системах программирования вводится специальный тип данных, определяющий структуру семафора, например, SEMAPHOR,SIGNAL,GETA.

Использование семафоров

```
begin  integer S; S:=1;
  parbegin
  task1: begin
                do while (true)
                P(S);
                <критический участок 1>;
                V(S);
                <обычный участок>;
                enddo
            end;
  task2: begin
                do while (true)
                P(S);
                <критический участок 2>;
```

```
                    V(S);
                    <обычный участок>;
                    enddo
                end
    parend
  end
```

54. Упорядоченные секции. Распараллелить цикл, используя упорядоченные секции и семафоры:

```
DO I=2,N
A(I) = 2*A(I) + C(I)
B(I) = C(I)*sin(X(I))
Y(I) = Y(I-1) + X(I)
D(I) = P(I)*B(I)/A(I)
ENDDO
```

Иногда, почти полностью распараллеливающийся по данным, алгоритм имеет единственную зависимость данных, которая на векторных машинах может потребовать расщепления цикла, чтобы изолировать зависимость.

Например:

```
      DO I = 1,N
       A(I) = EXP(SIN(B(I))) * EXP(COS(B(I)))
       C(I) = (D(I) + E(I)) / F(I) * A(I)
       X(I) = S * X(I-1) + C(I)
       Z(I) = X(I) * Y(I) + EXP(Z(I))
      ENDDO
```

Оператор присваивания X(I) включает рекурсию, что запрещает векторизацию. Обычно либо компилятор, либо программист выделит этот оператор в отдельный скалярный цикл, чтобы позволить другим трем операторам присваивания выполняться в параллельном режиме.

Это не всегда необходимо делать при распараллеливании. Если имеется достаточно работы выше и/или ниже оператора присваивания X(I) и N достаточно велико, можно достичь эффективного параллельного выполнения цикла путем помещения оператора присваивания X(I) в упорядоченную критическую секцию. Она будет обеспечивать доступ только одной нити в один и тот же момент явным или неявным замковым (lock) механизмом. В данном случае требуется выполнять итерации в таком порядке, чтобы удовлетворить требования рекурсии для вычисления X.

Реальная система синхронизации для SEQ

```
CALL SETSEQ (SEQ,0)
* инициализация, установка SEQ=0
C$DIR DO_PARALLEL (ORDRED)
DO I = 1,N
      A(I) = EXP(SIN(B(I))) * EXP(COS(B(I)))
      C(I) = (D(I) + E(I)) / F(I) * A(I)
      CALL WAITSEQ (SEQ,I-1)
      * ждать, пока SEQ будет равно  I-1
      X(I) = S * X(I-1) + C(I)
      CALL POSTSEQ (SEQ,I)
      *  установить SEQ в I
      Z(I) = X(I) * Y(I) + EXP(Z(I))
ENDDO
```

Семантика (для случая четырех процессоров):

- Инициализируются вычисления первых четырех итераций в заданном порядке на четырех параллельных процессорах.
- Все четыре процессора вычисляют свои A(I),C(I) одновременно.
- Первый процессор выполнит для I=1 присваивание X(1), в то время как другие ждут.
- Первый процессор для I=1 вычисляет Z(1), в то время как X(2) вычисляется вторым процессором для I=2.
- Первый процессор начинает вычислять новую итерацию для I=5, в то время как другие три процессора проходят сквозь критическую секцию также упорядочено.

Теперь, если работа по выполнению цикла хорошо сбалансирована, первый процессор для I=5 достигает точки WAITSEQ как раз тогда, когда четвертый процессор для I=4 закончит POSTSEQ. С этой точки все процессоры работают без простоя, пока они не достигнут последние четыре итерации цикла.

55. Статический и динамический способы образования параллельных процессов .

"Процесс - группа ячеек памяти, содержимое которых меняется по определенным правилам. Эти правила описываются программой, которую интерпретирует процессор." /Цикритзис Д./.

Вычислительный процесс можно рассматривать как последовательность команд ЭВМ, которая работает с данными ей ресурсами. (Задача, задание, процесс, нить, TASK). Так, на микропроцессоре могут работать одновременно несколько независимых процессов: счет вычислительной задачи, редактирование текстов и т.д. Далее рассматриваются вычислительные процессы, одновременно выполняющиеся на нескольких процессорах для решения общей задачи. Работа такой задачи может начинаться с порождения главного процесса, который при работе может порождать другие процессы динамически. Эти процессы могут работать параллельно с главной и также порождать другие процессы.

Другим способом порождения процессов является статический способ, когда производится одновременная инициализация на всех процессорах одинаковых процессов (SPMD – Single Program Multiple Data). Эти процессы опрашивают состояние решающего поля и настраиваются на выполнение своей части вычислений.(Они могут узнать: сколько их порождено, свои уникальные, внутренние имена, и т.д.)

Процессы, работающие на равных правах (не вызовы процедур и не процессы, связанные понятиями "главная-подчиненная"), иногда называемые сопроцессами, могут выполняться параллельно и при этом общаться друг с другом - не часто (слабо связанные процессы).

56. Требования к системам программирования методом передачи сообщений.

57. Система программирования MPI.

Главные цели создания системы параллельного программирования:

- Создать интерфейс прикладного программирования для MPP систем;
- Обеспечить возможность эффективных коммуникаций для коммуникации точка-точка, коллективных операций, группы процессов.
- Допускать удобное сопряжение с языками C, Fortran 77, Fortran 90 и C++;
- Простой способ создания процессов для модели SPMD (одна программа используется для обработки разных данных на разных процессорах);

Основные понятия языка

Группа - упорядоченное (от 0 до ранга группы) множество идентификаторов процессов Группы служат для указания адресата при посылке сообщений (процесс-адресат специфицируется своим номером в группе), определяют исполнителей коллективных операций. Являются мощным средством функционального распараллеливания - позволяют разделить группу процессов на несколько подгрупп, каждая из которых должна выполнять свою параллельную процедуру. При этом существенно упрощается проблема адресации при использовании параллельных процедур.

Контекст - область "видимости" для сообщений, аналогичное области видимости переменных в случае вложенных вызовов процедур. Сообщения, посланные в некотором контексте, могут быть приняты только в этом же контексте. Контексты - также важные средства поддержки параллельных процедур.

Коммуникаторы - позволяют ограничить область видимости (жизни, определения) сообщений рамками некоторой группы процессов, т.е. могут рассматриваться как пара - группа и контекст. Кроме того, они служат и для целей оптимизации, храня необходимые для этого дополнительные объекты.

Имеются предопределенные коммуникаторы (точнее, создаваемые при инициализации MPI-системы): * MPI_COMM_ALL - все процессы и операции над группами (локальные, без обмена сообщениями), например, Дай

размер группы. MPI_GROUP_SIZE(IN group, OUT size) Дай номер в группе обратившегося процесса. MPI_GROUP_RANK(IN group, OUT rank)

Основные операции - send, receive

Операции могут быть блокирующими и неблокирующими.

В операции send задается:

- адрес буфера в памяти;
- количество посылаемых элементов;
- тип данных каждого элемента;
- номер процесса-адресата в его группе;
- тег сообщения;
- коммуникатор.

MPI_SEND(IN start, IN count, IN datatype, IN dest, IN tag, IN comm) Типы данных - свои в MPI, но имеется соответствие между ними и типами Fortran и C.

В операции receive задается:

- адрес буфера в памяти;
- количество принимаемых элементов;
- тип данных каждого элемента;
- номер процесса-отправителя в его группе;
- тег сообщения;
- коммуникатор;
- ссылка на объект-статус, содержащий необходимую информацию о состоянии и результате операции.

Имеется возможность указать "любой отправитель" и "любой тег".

Имеется три режима коммуникаций - стандартный, режим готовности и синхронный.

В стандартном режиме последовательность выдачи операций send и receive произвольна, операция send завершается тогда, когда сообщение изъято из буфера и он уже может использоваться процессом.

В режиме готовности операция send может быть выдана только после выдачи соответствующей операции receive, иначе программа считается ошибочной и результат ее работы неопределен. В синхронном режиме последовательность выдачи операций произвольна, но операция send завершается только после выдачи и начала выполнения операции receive. Во всех трех режимах операция receive завершается после получения сообщения в заданный пользователем буфер приема.

Неблокирующие операции не приостанавливают процесс до своего завершения , а возвращают ссылку на коммуникационный объект, позволяющий опрашивать состояние операции или дожидаться ее окончания. Имеются операции проверки поступающих процессу сообщений, без чтения их в буфер (например, для определения длины сообщения и запроса затем памяти под него).

MPI относится к классу систем параллельного программирования, основанного на расширении стандартного последовательного языка описания процессов, библиотечными функциями для обмена сообщениями между процессами. В настоящее время MPI становится стандартом de facto и вытесняет ранние версии библиотечных расширений последовательных языков для параллельного программирования, например, таких как p4, PVM,Express и PARMACS. MPI - сложная система. Полностью она включает 129 функций, многие из которых имеют разнообразные параметры или варианты. Однако ключевая концепция MPI сосредоточена примерно в наборе из 20-25 функций, которые обеспечивают более чем адекватную поддержку для широкого спектра приложений.

Основное назначение инструментальных средств MPI - дать пользователю средства для организации локальной, глобальной и асинхронной связи между процессами и обеспечить разработку модульных программ, как из последовательных, так и из параллельных компонентов.

В модели программирования MPI вычисление включает один или большее количество процессов (потоков команд), связь между которыми осуществляется сообщениями, инициируемыми вызовом библиотечных подпрограмм. В большинстве MPI реализациях набор процессов зафиксирован и создан на этапе инициализации программы по одному процессу на процессор. Однако эти процессы могут выполнять различные программы. Следовательно, модель программирования MPI относится к классу multiple program multiple data (MPMD) в отличие от SPMD модели (simple program multiple data), в которой все процессоры выполняет одну программу. Во многих реализациях MPI параллельных процессов может быть и больше, чем реальных процессоров в вычислительной системе, тогда в функции MPI входит и распределение процессов. Псевдопараллелизм обеспечивает операционная система.

Поскольку число процессов в MPI вычислениях обычно зафиксировано, центральными в идеологии MPI являются механизмы синхронизации и обмена данными между процессами. Операции связи между процессами можно подразделить на:

- *двухточечные*, необходимые для того, чтобы послать сообщение от одного именованного процесса к другому. Эти операции могут использоваться для обеспечения локальной и неструктурной связи.
- группа процессов может инициировать *коллективные* операции связи, обычно для того, чтобы выполнить внешние действия с глобальными переменными (операции типа суммирования или оповещения).
- MPI поддерживает *асинхронную связь*

Эффективность и надежность обеспечиваются:

- определением MPI операций не процедурно, а логически, т.е. внутренние механизмы выполнения операций скрыты от пользователя;
- использованием непрозрачных объектов в MPI (*группы*, *коммуникаторы*, *типы* и т.д.);
- хорошей реализацией функций передачи данных, адаптирующихся к структуре физической системы.
- Обменные функции разработаны с учетом архитектуры системы, например, для систем с распределенной памятью, систем с общей памятью, и некоторых других систем, что позволяет минимизировать время обмена данными.

Переносимость обеспечивается:

- *во-первых*, тем, что тот же самый исходный текст параллельной программы на MPI может быть выполнен на ряде машин (некоторая настройка необходима, чтобы взять преимущество из элементов каждой системы). Программный код может одинаково эффективно выполняться, как на параллельных компьютерах с распределенной памятью, так и на параллельных компьютерах с общей памятью. Он может выполняться на сети рабочих станций, или на наборе процессоров на отдельной рабочей станции.
- *Во-вторых*, переносимость обеспечивается способностью параллельных программ выполняться на гетерогенных системах, то есть на системах, состоящих из процессоров с различной архитектурой. MPI обеспечивает вычислительную модель, которая скрывает много архитектурных различий в работе процессоров. MPI автоматически делает любое необходимое преобразование данных и использует правильный протокол связи, посылается ли код сообщения между одинаковыми процессорами или между процессорами с различной архитектурой. MPI может настраиваться как на работу на однородной системе, так и на работу на гетерогенной системе.
- *В третьих*, такими механизмами, как: определение одного вычислительного компьютера в виде виртуального компьютера (см. секцию 2.1) и возможностью задания произвольного количества таких виртуальных компьютеров в системе не зависимо от количества физических компьютеров (зависимость только от объема оперативной памяти в системе).
- *В четвертых*, переносимость обеспечивается заданием виртуальных топологий (см. секцию 2.1). Отображение виртуальных топологий на физическую систему осуществляется системой MPI. Виртуальные топологии обеспечивают оптимальное приближение архитектуры системы к структурам задач при хорошей переносимости задач.
- *В пятых*, компиляторами для Fortran(a) и C.

Уровень языка параллельного программирования определяется языковыми конструкциями, с помощью которых создаются параллельные программы. Как было сказано выше, операторы задания топологий, обменов данными и т.п., нужно задавать в программе явно и по этому языковый уровень параллельной программы оказывается ниже уровня последовательной программы. Наличие в системе таких средств как: виртуальные топологии, коллективные взаимодействия, создаваемые пользователем типы данных и др., значительно повышают уровень параллельного программирования по сравнению с системами с передачей сообщений, у которых нет таких средств.

58. Средства описания и создания процессов в языке Фортран-GNS.

По семантике языка Fortran GNS все процессы программы получают уникальные имена - системные идентификаторы, по которым производится отождествление корреспондентов при обмене сообщениями между процессами и в терминах которых задается среда выполнения функций редукции. Эти имена вырабатываются функцией порождения процессов - встроенной функцией языка NEWTASK. Структура системного идентификатора определяется реализацией, это имя не имеет внешнего представления.

Для манипулирования именами процессов внутри программы в язык Fortran GNS вводится новый для Фортрана тип данных - тип TASKID. Значения этого типа представляют собой имена процессов, которые вырабатываются функцией порождения процессов.

При статической модели параллелизма системные идентификаторы процессов вырабатываются системой запуска программы и могут быть получены пользователем встроенными функциями.

Добавляется константа .NOTASKID., которая представляет собой "нулевое" значение для объектов этого типа. Данные типа TASKID должны быть описаны в операторе TASKID, синтаксис которого аналогичен синтаксису операторов объявления типа в Фортране 77. Данные этого типа могут использоваться в операторах EQUIVALENCE, но при этом могут быть эквивалентными только данным типа TASKID. Допускается использование TASKID в качестве описателя типа подпрограммы-функции.

Использование переменных данного типа в операторах COMMON оговаривается в инструкциях по использованию языка на конкретных установках, в них же приводятся размеры имени процесса для расчетов длин общих блоков. В реализации на базе Фортран МВК [3] допускается использование данных типа TASKID в общих блоках, их длина - одно слово (как у данных типа INTEGER).

Для данных типа TASKID определены следующие операции: операции .EQ. , .NE. и присваивание (когда операнды операций, левая и правая части операторов присваивания имеют тип TASKID).

Данные типа TASKID используются в операторах передачи/приема сообщений - SEND и RECEIVE для задания абонентов - получателя или отправителя . Такие данные могут также передаваться в качестве параметров процедурам и функциям, а также могут быть переданы другим процессам с помощью операторов передачи сообщений.

Значения типа TASKID не могут использоваться в списках ввода-вывода, так как не имеют внешнего представления.

Начальная инициализация переменных этого типа также, как и других переменных Фортрана, не предусматривается, поэтому использование данных до присваивания им значений некорректно. Присвоение начальных значений зтим данным в операторах DATA не предусмотрено.

Спецификация данных типа TASKID в виде массива, структура которого совпадает с конфигурацией вычислительной системы, облегчает программирование задачи, например, может унифицировать операторы передачи данных между соседними процессами. В языке есть возможность заданием идентификатора массива данных типа TASKID указать массовую операцию для всех его элементов: в функциях порождения процессов,редукции и операциях передачи сообщений. Такая запись может ускорить выполнение программы, так как такие массовые операции могут выполняться параллельно.

Для задания массовых операций над частью массива - подмассива можно присвоить подмассиву имя оператором EQUIVALENCE, динамическое формирование структур подмассивов можно производить операторами языка Фортран МВК. Например:

```
TASKID T1(100),T2(100),TM
DIMENSION TM(100,2)
EQUIVALENCE (TM(1,1),T1(1)),(TM(1,2),T2(1))
```

При статической модели параллелизма инициализация данных этого типа производится функциями преобразования номеров виртуальных процессоров в имена задач.

Задачи и процессы

В языке вводится дополнительный вид программных единиц - программные единицы-задачи. Программа может содержать (помимо традиционных для Фортрана программных единиц) одну или более программных единиц-задач.

Главная программная единица (main program) определяет главный и единственный - начальной процесс программы. Эта единица может иметь заголовок:

PROGRAM *n*, где *n* - имя программы.

По Фортрану, заголовок программы может отсутствовать.

Имя программы (MAIN при его отсутствии) идентифицирует главную программную единицу программы, ссылки на него невозможны. (Конкретные реализации могут разрешать использование этого имени в операторах передачи сообщений в качестве адресата.)

При запуске начального процесса ему присваивается другое имя - системный идентификатор, по которому на него могут ссылаться другие процессы программы при передаче сообщений или при выполнении функций редукции. Это имя может быть получено любым процессом при помощи встроенной функцией MASTER без параметров. Результат функции имеет тип TASKID.

Программная единица-задача являются описанием задачи. Вычислительные процессы создаются по описанию задачи динамически и во время создания им присваиваются уникальные имена - системные идентификаторы. По одному описанию задачи может быть образовано произвольное число процессов данного класса.

Первым оператором программной единицы-задачи является оператор заголовка, имеющий вид: TASK PROGRAM *n*, где *n* - имя программной единицы-задачи.

В остальном, структура программной единицы-задачи не отличается от структуры главной программной единицы Фортрана 77. В частности, последним оператором, как обычно, является оператор END.

Задачи не могут иметь общей памяти с другими задачами, в частности, с главной, поэтому общие блоки (COMMON-блоки) задач локальны в пределах каждой задачи и программных единицах, которые в ней используются. Семантика COMMON-блоков в пределах одной задачи - традиционная.

Создание процессов.

Для создания процессов в динамической модели параллелизма используется стандартная функция NEWTASK. Обращение к этой функции производится операторами присваивания вида:

*i*t = NEWTASK (*n, ia*) , где

n - имя программной единицы-задачи;

ia - константа, имя переменной, элемента массива или массива целого типа; значения этого параметра определяют виртуальные номера вычислителей, на которых создаются порождаемые процессы.

*i*t - имя переменной, элемента массива или массива типа TASKID, куда заносятся имена (системные идентификаторы) порожденных процессов.

Функция NEWTASK создает по описанию задачи с именем *n* вычислительные процессы, количество которых определяется числом виртуальных номеров вычислителей. Это число равно количеству элементов массива параметра *ia* или равно одному для скалярного параметра.

Результатом выполнения функции NEWTASK является имя процесса или массив имен (системных идентификаторов) созданных процессов. Эти имена присваиваются переменным *i*t.

Если *n* не совпадает с именем программной единицы-задачи, по которой образован текущий процесс, то оно должно быть описано. По аналогии с оператором EXTERNAL, для описания имен внешних задач вводится оператор спецификации TASK EXTERNAL, имеющий вид:

TASK EXTERNAL *n* [, *n*i]... где

*n, n*i - имена программных единиц-задач.

Имена, указанные в списке этого оператора, разрешается использовать в качестве фактических параметров процедур (в том числе в качестве параметров функций NNAME и NEWTASK) и в качестве адресатов в операторе SEND. Если в качестве таковых параметров используется имя текущей единицы - задачи, то оно также должно быть описано как внешнее. Таким образом, передать сообщение процессу можно, идентифицируя его именем программной единицы-задачи, по которой он был образован (по имени класса), или по системному имени. Оператор TASK EXTERNAL должен быть размещен в программе-задаче до первого исполняемого оператора. Имена (системные идентификаторы) процессов не имеют внешнего представления, они скрыты от пользователя, их можно присваивать переменным типа TASKID. Процесс может определить собственное имя, имя процесса, породившего его, и имя главного процесса программы с помощью стандартных функций MYTASKID, PARENT и MASTER. Определение имен процессов по номерам виртуальных процессоров приведено в п. 7.

Созданные процессы загружаются и сразу начинают выполняться на процессорах, виртуальные (логические) номера которых заданы значениями параметра *ia*. Допускается использование одинаковых логических номеров для размещения процессов на одном виртуальном процессоре (для статической модели параллелизма это не так !).

Конкретное отображение виртуальных номеров процессоров на процессоры рабочей машины, способ загрузки процессов определяется реализацией, например, конструкциями языка описания конфигурации (аналогично тому, как номера каналов ввода-вывода в стандартах Фортрана не закреплены, а определяются реализацией, в частности, языком управления заданиями). Процедура создания главного процесса программы и порядок его размещения на процессоре также определяется реализацией.

При реализации статической модели параллелизма распределение процессов по процессорам производиться статически, на этапе подготовки задания. В этом случае, функции NEWTASK в программе отсутствуют и процессы запускаются при инициализации задачи. В статическом варианте имена процессов (системные идентификаторы) определяются с помощью функции GETTASKID, имеются еще ряд стандартных функций для работы в этом режиме (см. 7.).

Примеры использования функций порождения процессов

Операторы порождения будут иметь окружение программы-задачи C.

```
      TASK PROGRAM C
      TASK EXTERNAL A,B,READ,PRINT
      TASKID TI, TM(100),TM2(100,2)
      INTEGER K,KM(100),KM2(100,2)
      .............
      END
```

Инициализация процессов:

```
1     TI = NEWTASK(A,K)
2     TM = NEWTASK(A,KM)
3     TM2 = NEWTASK(A,KM2)
      DO 4 I=1,100
4     TM(I) = NEWTASK(A,KM(I))
```

```
      DO 5 I=1,100
      TM2(I,1) = NEWTASK(A,KM2(I,1))
5     TM2(I,2) = NEWTASK(B,KM2(I,2))
```

Операторы с метками 2 и 4 (3 и 5) производят одинаковые действия, но инициализация в форме 2 (3) предпочтительнее, так как эта запись позволяет производить инициализацию процессов параллельно.

Отсутствие в Фортране 77 аппарата вырезок из массивов не позволяет иметь такую запись для алгоритмов:

```
      DO 6 I=1,99,2
      TM(I) = NEWTASK(A,KM(I))
6     TM(I+1) = NEWTASK(B,KM(I+1))
```

или

```
      TM(1) = NEWTASK(READ,KM(1))
      TM(100) = NEWTASK(PRINT,KM(100))
      DO 7 I=2,99
7     TM(I) = NEWTASK(B,KM(I))
```

В результате выполнения операторов: TI = NEWTASK(PRINT,1) 8 TI = NEWTASK(READ,2)) на виртуальном процессоре с номером 1 будет образован и запущен процесс PRINT, внутреннее имя которого потеряно и недоступно программе из-за выполнения оператора 8.Доступ к нему для передачи сообщений возможен только по программному имени, заказать от него прием сообщений невозможно. С процессом на втором процессоре возможен обмен сообщениями в обе стороны.

В языке определена только одна синтаксическая конструкция использования функции NEWTASK - в операторе присваивания. Тем не менее входные языки могут разрешать использование этой функции и в других контекстах, там, где употребляются переменные типа TASKID. Например, оператор:
SEND (NEWTASK(PRINT,MI)) RES породив процесс печати, передает ему данные через RES.

Виртуальные номера процессоров.

Кодировка виртуальных номеров процессоров произвольная, также как задание номера меток программы. Эти номера могут быть переданы процессу, который порождает другие, через каналы ввода, операторами передачи сообщений, а также сгенерированы программно.

Одинаковые номера процессоров могут быть использованы для оптимального распределения ресурсов путем выравнивания времени работы процессов. Определив время работы (число операций) самого время-емкого процесса, можно объединять на одном процессоре выполнение нескольких других процессов, если их суммарное время меньше.При этом можно ускорить выполнение программы из-за упрощения коммуникаций между процессами, выполняемых на одном процессоре.

Виртуальные номера процессоров могут быть использованы для настройки решающего поля. Параллельные алгоритмы, формулируемые в теоретической постановке обычно как n-размерные, при программировании приводятся к параметрам реальной структуры решающего поля. Алгоритмы, реализуемые как стандартные процедуры, также должны адаптироваться к структуре выделяемых ресурсов. Настройка структуры (под) программ под параметры вычислительной среды может управляться заданием номеров виртуальных вычислителей.

При запуске программы на конкретной вычислительной платформе возможно потребуется сопроводить программу спецификацией об используемых значениях номеров виртуальных процессоров и схемой их оптимальной дислокации на вычислителях платформы, и об именах задач, связанных с этими номерами для последующей загрузки соответствующих процессов. Конкретные реализации, оговаривая правила загрузки задач на процессоры, могут содержать средства задания порядка выполнения (приоритета) нескольких процессов на одном процессоре.

Для статической модели параллелизма балансировка загрузки вычислительной системы может производиться на уровне языка конфигурации, размещением на одном физическом процессоре нескольких виртуальных.

59. Средства передачи и приема сообщений в языке Фортран-GNS.

Обмен информацией между процессами осуществляется с помощью передачи сообщений. Для этого используются операторы:

SEND - послать сообщение и RECEIVE - принять сообщение.

Предусмотрено три способа передачи сообщений: синхронный, асинхронный и передача без ожидания.

Синхронный способ

При синхронном способе передачи сообщений посылающий и принимающий процессы приостанавливают выполнение своих программ и переходят в состояние ожидания до тех пор, пока не выполнятся оба синхронных оператора SEND и RECEIVE.

Сообщения данного типа могут передаваться без использования буферов почтовых ящиков, так как передача синхронного сообщения будет производиться только при готовности абонентов к обмену и может происходить непосредственно из памяти передающего процесса в память принимающего процесса.

Процесс может посылать синхронные сообщения как отдельному процессу, так и группе процессов. В последнем случае процесс отправитель продолжит выполнение только после получения подтверждения о выполнении соответствующих операторов RECEIVE во всех процессах - получателях. Получив сообщение, получатель группового синхронного сообщения может продолжить работу, не ожидая завершения всех остальных обменов по данному оператору SEND. Получатель синхронного сообщения может ждать его только от процесса, имя которого указано в параметре оператора RECEIVE, т.е нельзя заказать в одном операторе ожидание синхронных сообщений от нескольких процессов.

Асинхронный способ

При асинхронном способе передачи сообщения посылающий процесс продолжает выполнение независимо от того, выполнил ли получающий процесс оператор RECEIVE или нет.

Сразу же после выполнения оператора SEND процесс может изменить значения данных, перечисленных в списке передаваемых значений этого оператора, однако абоненту передаются те значения, которые были на момент выполнения оператора SEND.

Выполнение асинхронного оператора RECEIVE и продолжение работы процесса - получателя производиться только после поступления сообщения. Если сообщение пришло к получателю до выполнения у него оператора RECEIVE, то сообщение помещается в почтовый ящик процесса без прерывания его выполнения. Сообщения с одинаковым значением тега от одного и того же отправителя помещаются в почтовом ящике в том же порядке, как они посылались и в таком же порядке они будут выбраны операторами RECEIVE.

Сообщения, передаваемые асинхронным способом, идентифицируются именами отправителя и тегами (положительными значениями целого типа). Имя отправителя можно опускать в процессе - получателе, т.е. можно заказывать ожидание сообщений этого типа (с заданным тегом) от произвольного процесса. В этом случае отправителя можно идентифицировать при получения сообщения при помощи параметра SENDER в операторе RECEIVE.

Стандартные функции TESTMSG, TESTTAG, PROBE позволяют анализировать наличие сообщений, значения тегов и имен отправителей асинхронных сообщений в почтовом ящике процесса и, соответственно, управлять порядком выбора поступивших сообщений.

Для организации селекции и установки режимов ожидания заданных асинхронных сообщений можно использовать также конструкции выбора (см. раздел 5.).

Передача без ожидания

При передаче сообщения способом "без ожидания" момент передачи сообщений (копирование данных для передачи в процессе отправителе и присваивание соответствующим переменным значений данных из принятого сообщения в процессе получателе) не определен в языке и зависит от системы интерпретации. При такой передаче работа обоих процессов (отправителя и получателя) продолжается после выполнения операторов этого вида, независимо от того, выполнилась ли фактически передача сообщения.

Так как значения данных, указанных в списке переменных оператора SEND, могут быть изменены любым следующим оператором процесса, то передаются такие значения, которые имеют эти переменные в момент фактической передачи. Оператор приема сообщений данного типа формирует заявку на прием сообщения, а переменные, перечисленные в списке не изменяются, если сообщение еще не поступило в почтовый ящик процесса на момент выполнения оператора. При поступлении сообщения, данные пересылаются из почтового ящика в память процесса (присваиваются соответствующим переменным) без прерывания его работы.

В каждом операторе обмена этого типа один из его параметров - логическая переменная FLAG служит для фиксации факта передачи сообщения. Значение этой логической переменной-флага связывается с фактом выполнения передачи данных, заданной оператором обмена для того процесса, в котором определена эта переменная. Система интерпретации присваивает флагу в процессе отправителе значение .TRUE. в момент снятия копии с данных отправителя для передачи, и флагу в процессе-получателе после присвоения этих значений переменным получателя. До выполнения оператора передачи / приема сообщения значение логической переменной - флага, связанной с этим оператором не определено.

Определить факт передачи данных и приостановить выполнение процессов до выполнения фактической передачи сообщения этого типа (для отправителя - до снятия копии с передаваемых данных, для получателя - до поступления сообщения в почтовый ящик или записи переданных данных в поле задачи) можно при помощи стандартных процедур - MSGDONE, ALLDONE, TESTFLAG.

Только установка значения флага в .TRUE. отправителю дает возможность изменять переменные, перечисленные в операторе SEND, без риска изменить передаваемые данные, для получателя - начать использовать полученные данные.Использование этих логических переменных вне операторов передачи сообщений и перечисленных выше процедур не допускается.

По аналогии с асинхронным протоколом допускается ожидание асинхронных сообщений от произвольного процесса (в операторах RECEIVE можно опускать имя отправителя).

Замечание:

Понятие "почтовый ящик процесса", используемое в данном описание языка, носит методический характер и служит моделью для пояснения семантических понятий аппарата передачи сообщений. Алгоритмы реализация этого аппарата зависят от особенностей вычислительной платформы и ее системного обеспечения.

60. Протоколы передачи и приема сообщений в языке Фортран-GNS.

Операторы SEND и RECEIVE имеют вид:

```
SEND (sm [,ss]...) [список] RECEIVE (rm
[,rs]...) [список]
```

где

sm или *rm* - спецификация сообщения - определяет адресата (процесс или процессы, которым посылается сообщение) или отправителя и способ синхронизации;

список - список передаваемых данных имеет такой же вид, как списки в операторах ввода/вывода Фортрана 77, т.е. элементом списка может быть константа,имя переменной, переменная с индексом, имя массива или неявный цикл;

ss или *rs* - дополнительная спецификация.

Вид спецификации сообщения *sm* и *rm* зависит от способа синхронизации.

а) Синхронный способ: *sm* **есть** [TASKID =] *адресат*,

где *адресат* **есть** *adr* **или** ALL, а

adr - ссылка на функцию, имя переменной, элемента массива или имя массива типа TASKID или имя программной единицы-задачи.

Если *adr* - имя программной единицы-задачи, то сообщение посылается всем процессам программы, образованным по образцу указанной программной единицы (исключая посылающую).процедуры.

ALL - означает, что сообщение посылается всем процессам программы, образованным на момент выполнения оператора передачи сообщения,исключая процесс-отправитель. *rm* **есть** [TASKID =] *t*, где *t* - переменная или элемент масси-
ва типа TASKID; параметр специфицирует процесс - отправитель.

Когда *adr* (*rm*) - значение типа TASKID, оно должно ссылаться на незавершенный процесс.

б) Асинхронный способ

sm **есть** [TAG=] *ie*, [TASKID =] *адресат*
или TASKID = *адресат*, [TAG =] *ie*
rm **есть** [TAG=] *ie* [,[TASKID =] *t*]
или TASKID = *t*, [TAG =] *ie*

где адресат и *t* определяются как *rm* для синхронного способа,

ie - выражение целого типа, значение которого определяет тег сообщения.

в) Способ без ожидания

sm **есть** [TASKID =] *адресат*, FLAG = *l*
rm **есть** [TASKID =] *adr*, FLAG = *l*
или [FLAG =] l

где

адресат и *adr* определяется как и для предыдущего способа;

l - имя логической переменной.

Переменная *l* может использоваться также в стандартных процедурах MSGDONE(*l*) и TESTFLAG(*l*) (см. п.7.). В других ситуациях ее использование не допускается.

Дополнительная спецификация

Дополнительная спецификация *ss* и *rs* , как видно из синтаксиса, является необязательной.

ss **есть** ERR = *l*
и / **или** SRSTAT = *ios*

где

l - специфицирует метку, которой передается управление в случае ошибки при передаче сообщения;

ios - имя переменной или элемента массива целого типа; специфицирует статус состояния (аналог спецификации IOSTAT в операторах ввода/вывода).

В результате выполнения оператора, переменной ios присваивается значение О , если не было обнаружено ошибки, и положительное значение, если обнаружена ошибка. Классификация видов ошибок определяется реализацией языка. Если спецификация SRSTAT отсутствует, то значение статуса состояния присваивается системной переменной.

Если спецификация ERR отсутствует, то в случае ошибки, задача (и вся программа) завершается выполнением стандартной процедуры ABORT.

В качестве дополнительной спецификации *rs* для оператора RECEIVE, помимо аналогичных спецификаций ERR и SRSTAT, можно также использовать необязательную спецификацию SENDER, т.е.

rs **есть** ERR = *l*

и / **или** SRSTAT = *i*

и / **или** SENDER = *t*

где t - переменная типа TASKID.

Спецификацию SENDER полезно использовать, если отсутствует имя процесса-отправителя. Переменной *t* после выполнения оператора RECEIVE присваивается значение имени процесса-отправителя.

Замечание.

1.Способ передачи сообщения и число элементов в списках оператора SEND и соответствующего оператора RECEIVE должны совпадать; для каждого элемента - тип, а для массивов также размерность и конфигурация должны соответствовать друг другу. Конфигурация массива - это одномерный массив целого типа, размер которого равен размерности исходного массива, а элементы равны размерам соответствующих измерений.

Несоответствие структуры данных при синхронной передаче считается ошибкой в обоих процессах , при других способах передачи - ошибкой в задаче-отправителе.

2. Если значение переменной адресата есть .NOTASKID. или ссылка на завершенный процесс, оператор обмена для этого процесса не выполняется, а спецификация SENDER получает значение .NOTASKID. .

3.Не считается ошибкой наличие в почтовом ящике невостребованных сообщений при завершении процесса.

Использование операторов передачи сообщений

Синхронный режим передачи сообщений

По определению, синхронный режим передачи сообщений требует одновременного выполнения в процессах соответствующих операторов, что при рассогласовании работы процессов может быть источником "зависания" программы. Достоинством данного метода передачи сообщений является принципиальная возможность проводить обмен данными без использования системных буферов, что может ускорить время обмена.

Операторы передачи сообщений с пустым списком данных могут использоваться для синхронизации вычислений программы.

Для этого в процессе, управляющем синхронизацией, следует выполнить оператор: SEND(ALL), во всех процессах программы - RECEIVE (T1),где T1 должен ссылаться на управляющую задачу. Управляющий процесс продолжиться только после выполнения во всех задачах оператора RECEIVE. Оператор SEND(A) требует синхронизации только от процессов, образованных по программе-задаче A, а SEND(TM) - от процессов, на которые ссылаются элементы массива TM. Если процессам известно, что рандеву будет требовать начальный процесс, то их синхронизирующий оператор может быть записан: RECEIVE (MASTER()). Точная синтаксическая запись: T1=MASTER()

RECEIVE (T1)

так как в параметрах операторов передачи сообщений не разрешается использования выражений. Реализации могут иметь расширенное толкование семантики, в частности, допускать использование функций типа TASKID в позициях переменных этого типа. Некоторые реализации могут запрещать в операторах приема сообщений неявные циклы, управляемые параметром, находящимся в этом же списке ввода. Так, в Фортране МВК запрещены такие конструкции: RECEIVE (<..>) N, (KM(I),I=1,N)

Асинхронный режим передачи сообщений

Использование данного способа обмена сообщениями делает программу еще менее критичной к согласованию операторов обмена сообщениями, так как процесс-отправитель продолжает работу после передачи сообщения, не дожидаясь конца фактической передачи сообщения (и даже начала, так как система интерпретации "должна" сразу же, скопировав передаваемые данные в буфер, "отпустить" процесс) . Получатель сообщений этого типа может выдавать директиву приема сообщения, только удостоверившись в наличии нужного сообщения в почтовом ящике процесса при помощи функций: TESTMSG,TESTTAG,PROBE. Выполнение оператора RECEIVE без проверки наличия сообщения в почтовом ящике процесса, по аналогии с синхронным способом обмена, приводит к задержки выполнения процесса до приема сообщения с заказанным тегом и ,возможно, с заданным именем отправителя. Но в отличие от синхронного способа, асинхронный способ позволяют проводить селекцию поступающих сообщений при помощи конструкций выбора. Пусть процесс может получать сообщения с тегом 1, но от процесса TI - скаляр целого типа, а от процесса TM(1) - массив из 100 целых чисел. Тогда прием таких сообщений может быть запрограммирован так:

```
IF (TESTTAG(1)) THEN
SELECT MESSAGE
CASE(1,TI)
RECEIVE (1) K
CASE(1,TM(1))
RECEIVE (1) KM
END SELECT
```

END IF или:

```
SELECT MESSAGE
CASE(1,TI)
RECEIVE (1) K
CASE(1,TM(1))
RECEIVE (1) KM
CASE DEFAULT
GO TO 2
END SELECT 2     CONTINUE
```

Ожидание этих сообщений, то есть, по аналогии с синхронной передачей сообщений, прерывание работы процесса до получения сообщения (в данном случае любого из ожидаемого) записывается так:

```
SELECT MESSAGE
CASE(1,TI)
RECEIVE (1) K
CASE(1,TM(1))
RECEIVE (1) KM
END SELECT
```

Если сообщения различаются тегами, то конструкция их ожидания может иметь вид:

```
SELECT MESSAGE
CASE(1)
RECEIVE (1) K
CASE(2,TM(1))
RECEIVE (2) KM
END SELECT
```

Режим передачи сообщений без ожидания

Передачи сообщений этого типа рекомендуется использовать для передачи данных "впрок", заблаговременно. Если после операторов SEND / RECEIVE без ожидания, поместить операторы: CALL MSDONE(L), семантика операторов будет совпадать с семантикой асинхронных операторов. Следует иметь ввиду, что при использовании операторов данного вида в цикле без использования процедур ожидания фактической передачи сообщений возможно искажение передаваемых данных; семантика функции TESTFLAG при этом будет двусмысленной.

Структура элементов списка передаваемых сообщений

Структура элементов списка передаваемых (принимаемых) сообщений, как объявлено, совпадает с списком ввода вывода операторов обмена Фортрана. Широкие возможности предоставляет аппарат неявных циклов.

Пусть имеются описания: REAL A(10),B(5,5). Тогда оператор:

SEND(ALL)A,(B(I),I=1,5),(B(I,1),I=1,5),(B(I,I),I=1,5) перешлет всем процессам программы следующие объекты: - массив А целиком, - первый столбец массива В, - диагональные элементы массива В.

Этим способом можно организовать не только передачу (прием) любых вырезок их массивов, но и множества одинаковых данных, например, переслать пять копий массива В: SEND(ALL)(B,I=1,5)

61. Идентификация абонентов при передачи сообщений в языке Фортран-GNS.

Идентификация абонентов при организации передачи сообщений может производится по имени процесса (массиву имен) или идентификатору задачи, по описанию которой образован процесс. Есть способы безадресных (широковещательных) посылок сообщений - "всем", и приема от произвольного отправителя сообщений с заказанной "маркой" - тегом. Идентификация абонентов может быть произведена:

По имени процесса

Основным способом идентификации абонентов при передаче сообщений является задание имени процесса (системного идентификатора) переменными типа TASKID и стандартными функциями этого типа MYTASKID, PARENT и MASTER. Использование этих стандартных функций является единственным первоначальным способом сослаться из порождаемых процессов на имена порождающих процессов, так как начальный диалог между процессами возможен только на уровне порожденный-порождаемый. Путем обмена сообщениями можно организовать передачу имен процессов для организации произвольной абонентской сети.

Переменные типа TASKID - параметры операторов передачи сообщений могут:

- ссылаться на функционирующий процесс,

- ссылаться на завершенный процесс,
- иметь значение .NOTASKID.,
- быть не инициализированными.

Реакция программы на три последних случая не оговаривается в языке и определяется реализацией. Одно из решений - игнорирование оператора обмена сообщениями с такими параметрами (кроме синхронных операторов, для которых такой случай следует считать "авостом" в программе). Процесс-отправитель имеет возможность организовать дублирование передаваемых сообщений для передачи их одновременно нескольким процессам-получателем. Для этого допускается задание абонентов в параметрах оператора передачи сообщений в виде массива типа TASKID. Сообщение будет передано при этом всем процессам, на которые есть ссылки через элементы массива. Данная массовая операция предпочтительнее передачи сообщений, организованной поэлементной циклической конструкцией. Цикл предписывает порядок перебора элементов массива - абонентов передачи, а для синхронного обмена и завершение очередной передачи до посылки следующего сообщения. Массовая операция обмена может проводиться параллельно и независимо для всех заказанных абонентов. Массовые операции ассиметричны, нельзя заказать ожидание сообщения от нескольких процессов сразу. Теговая идентификация сообщений и конструкции выбора допускает некоторую вольность в указаниях об именах отправителей.

По имени программной единицы-задачи

Задание в операторе передачи сообщений в качестве адресата - получателя имени программной единицы-задачи означает рассылку сообщений всем процессам, образованным по описанию указанной задачи на момент выполнения оператора. Случай, когда к этому моменту не было образовано ни одного процесса по указанному образцу, можно рассматривать по аналогии с передачей сообщения процессу с "именем" - .NOTASKID. . Возможность задать в качестве адресата имя главного процесса уточняется в описаниях входных версий языка.

Безымянные абоненты

Специальный идентификатор ALL в параметре адресата-получателя сообщения предписывает передать данное сообщение всем процессам программы, инициализированным на данный момент.

Передача сообщений двумя последними способами предполагает возможность параллельной рассылки сообщений и исключение процесса, выполняющего оператор передачи сообщения, из списка абонентов - получателей.

Помеченные сообщения

При передаче сообщений асинхронным способом сообщения при отправлении помечаются целым числом - тегом. Эта разметка дает возможность абонентам идентифицировать сообщения от разных операторов передачи сообщений одного процесса. Получив сообщение с заданным тегом, процесс может узнать имя отправителя, через аппарат SENDERов. Это замечание верно и для передачи сообщений без ожидания.

62. Автоматическое распараллеливание последовательных программ.

http://parallel.ru/tech/tech_dev/auto_par.html

63. Семантика циклов, выполняемых параллельно на ОКМД системах.

64. Алгоритмы преобразования программ методом координат.

Одним из методов распараллеливания программ является метод координат, обеспечивающий векторизацию циклов для синхронных ЭВМ (ОКМД - SIMD архитектуры). Семантика синхронных циклов следующая: для каждой итерации цикла назначается процессор; вычисления должны производиться всеми процессорами (процессорными элементами) параллельно и синхронно. Операторы тела цикла выполняются по очереди, но каждый оператор выполняется всеми процессорами одновременно. При выполнении оператора присваивания сначала вычисляется правая часть, затем одновременно выполняется операция присваивания для всех элементов массива левой части.

Теоретическое обоснование классического метода координат предложено Лэмпортом. Метод применим для канонических ("чистых") циклов, тела которых состоят только из операторов присваивания, причем управляющая переменная заголовка цикла (параметр цикла) должна входить в индексные выражения операторов тела цикла

линейно. Метод предназначен для синхронной векторизации одномерных циклов (многомерные циклы можно рассматривать по каждому параметру отдельно). Идея данного метода состоит в том, что для индексов, содержащих параметры цикла, рассматривается порядок следования, который и определяет очередность выборки/записи соответствующего элемента массива. Так как рассматриваются только линейные индексные выражения, порядок следования определяется значениями соответствующих алгебраических выражений. Строится граф характеристических множеств всех индексов, и при отсутствии в графе петель делается заключение о возможности векторизации всего цикла.

В некоторых случаях (устранимая векторная зависимость) векторизация возможна лишь в случае переупорядочивания - перестановки операторов цикла или путем введения временной переменной (массива), куда будут копироваться элементы вектора для защиты перед их изменением для последующего использования в исходном виде. Итак, метод координат:

- позволяет определить возможность выполнения цикла в синхронном режиме;
- содержит алгоритмы преобразования тела цикла к синхронному виду.

Например, по Лэмпорту, цикл:

```
      DO 24 I=2,M
      DO 24 J=2,N
21    A(I,J) = B(I,J)+C(I)
22    C(I) = B(I-1,J)
23    B(I,J) = A(I+1,J) ** 2
24    CONTINUE
```

преобразуется в цикл:

```
      DO 24 J=2,N
      DO 24  SIM FOR ALL I {i:2<=i<=M}
C  SIM - SI Multeneusly (одновременно)
      TEMP(I) = A(I+1,J)
21    A(I,J) = B(I,J)+C(I)
23    B(I,J) = TEMP(I) ** 2
22    C(I) = B(I-1,J)
24    CONTINUE
```

Конструктивный алгоритм реализации метода координат.

Пусть, цикл одномерный, все операторы тела - операторы присваивания, в левых частях которых - переменные с индексом - параметром цикла. Индексы линейные, т.е. имеют вид i+/-b,где b - константа. Определяются отношения следования для всех пар одноименных переменных с индексами: генерируемых переменных (переменные в левой части операторов присваивания - f) и используемых переменных (входящих в формулы правых частей - q). Отношение следования для одной пары зависит от порядка обращения к одному и тому же элементу массива при выполнении цикла. Так, для A(I) = A(I+1) отношение можно кодировать: f<q . Для многомерных циклов для всех пар (f,q) определяются направляющие вектора, элементы которых состоят из символов (=, >, <) в позиции соответствующего индекса, которая определяется порядком индексов в гнезде цикла. В примере Лемпорта (исходном) для A вектор имеет вид: (<,=).

Для тела цикла, состоящего из одного оператора, отношения, и соответственно, семантика выполнения, могут быть: < > = и их сочетание. Например, тело цикла: A(I) = (A(I)+A(I+1))/2 допускает векторное выполнение; оно невозможно при наличии в операторе хотя бы одного соотношения: f>q. Для тела цикла: A(I) = A(I+C))/2 при C>0 возможно синхронное выполнение, при C=0 кроме синхронного, возможно параллельное асинхронное; а при C<0 параллельное выполнение невозможно.

Для многомерных (тесногнездовых) циклов такое исследование может проводиться для каждого параметра, а для распараллеливания выбираться наиболее оптимальный вариант. Для самого внутреннего цикла определение следования можно проводить, считая его одномерным. Для любого другого цикла, исследование можно проводить по такой же схеме, если семантика цикла допускает перестановку его на позицию самого внутреннего. Перестановка допустима, если для всех новых направляющих векторов крайне левый, отличный от = , элемент сохранил направление. Например, для двумерного цикла, наличие вектора (<,>) делает перестановку некорректной. Так для цикла:

```
   DO 99 J=2,M
   DO 99 K=2,M
99  U(J,K) = (U(J+1,K)+U(J,K+1)+U(J-1,K)+U(J,K-1)) .25
```

направляющие вектора:

1 <U(J,K),U(J+1,K)> = (<,=)
2 <U(J,K),U(J,K+1)> = (=,<)
3 <U(J,K),U(J-1,K)> = (>,=)

2 <U(J,K),U(J,K-1)> = (=,>)

Отсюда:

- операторы циклов по I и J можно менять местами,
- часть оператора, включающая вхождения из векторов 1 и 2, можно вынести в отдельный оператор, который векторизуется как по I так и по J.

Для двух операторов тела цикла отношения следования переменных с индексами можно систематизировать и кодировать так (qi в общем случае, список - информационные поля q и f) : A - независимые отношения B - (qi>qj) , C - (qi<qj) , K - (qi=qj) G - B+K,B+C,B+K+C , H - неоднозначные отношения. Тогда по координатам приводимой ниже таблице можно определить типы связей и метод выполнения цикла для этих операторов. Если объединить информационные поля двух операторов , то таблицу можно использовать для анализа третьего оператора и т.д.

```
*   Таблица решений для метода координат
* Оператор 1      O1 = I1   Oi,Ii - список переменных с индексами
* Оператор 2      O2 = I2     ->  - порядок анализа отношений
*-------------------------------------------------------------------
*             .                    O2 -> I1                        .
*             .    A    .    B    .    K    .    C    .   G   .   H  .
*             .  НЕЗАВ. .  O > I  .  O = I  .  O < I  . O <=>I   *   .
* I2 -> O1    .         .         .         .         .       .      .
*-------------------------------------------------------------------
*             .         .         .         .         .       .      .
*   A  НЕЗАВ. .    P    .    R    .    S    .    M    . R(K,C).   T   .
*
*_____________________________________________________________________
*             .         .         .         .         .       .      .
*   B  I > O  .    R    .    R    .  M(B)   .  M(B)   . R(K,C).   T   .
*
*_____________________________________________________________________
*             .         .         .         .         .       .      .
*   K  I = O  .    S    .    T    .   S     .   M     .   T   .   T   .
*_____________________________________________________________________
*             .         .         .         .         .       .      .
*   C  I < O  .    M    .    T    .   M     .   M     .   T   .   T   .
*_____________________________________________________________________
*             .         .         .         .         .       .      .
*   G  I <=> O.  M(B)   .    T    .  M(B)   .  M(B)   .   T   .   T   .
*_____________________________________________________________________
*             .         .         .         .         .       .      .
*   H    *    .    T    .    T    .    T    .    T    .   T   .   T   .
*_____________________________________________________________________
*   Кодировка отношений следования
*    A - независимые отношения
*    B - (qi>qj) , C - (qi<qj) , K - (qi=qj)
*    G - B+K,B+C,B+K+C
*    H - неоднозначные отношения
*  ТИПЫ  СВЯЗЕЙ
*     P - независимые операторы
*     M - SIMD , M(EA) - SIMD с защитой  EA
*     R - SIMD с реверсом, R(EA) - и с защитой EA
*     S - PAR (асинхронная параллельность)
*     T - запрет векторизации
* Защита EA - копирование массивов EA перед
```

В общем случае, алгоритм векторизации методом координат с использования данной таблицы следующий.

Обработка тела цикла начинается с анализа на возможность векторного выполнения каждого оператора тела в отдельности. Затем к первому оператору добавляется второй и проводится анализ на возможную их векторизацию. Пара получает тип, определяющий возможность векторизации ее компонент, информационные поля операторов сливаются, и на следующем шаге применения процедуры векторизации пара рассматривается как единый супероператор. Таким образом из всех операторов тела цикла образуется один супероператор и, если его тип есть T, то векторизация тела цикла невозможна. Для всех других типов производится обратный анализ полученного графа (супероператора). Если при этом связки имели тип R или R(EA),M(EA), то хотя они и допускают асинхронное параллельное выполнение, но необходимы преобразование тела цикла. Связки типа T дают операторы, векторизация которых невозможна. Интерпретация остальных типов связок очевидна. В процессе формирования супероператоров к связкам типа T могут применяться процедуры поиска минимальных границ области типа T и чистки области T.

Примеры векторизации

Исходные тела циклов	Преобразованные тела циклов
A(I) = B(I) C(I) = A(I+1)	C(I) = A(I+1) A(I) = B(I)
A(I) = B(I)	D(I) = A(I)

```
C(I) = A(I) + A(I+1)          A(I) = B(I)
                              C(I) = A(I) + D(I+1)

A(I) = B(I) + B(I+1)          D(I) = B(I)
B(I) =  A(I+1)                B(I) = A(I+1)
                              A(I) = D(I) + B(I+1)

A(I) = B(I) + B(I-)            Векторизация невозможна
B(I) =  A(I)
```

65. Схема преобразования программ методом гиперплоскостей.

Метод гиперплоскостей применим только к многомерным циклам. В пространстве итераций ищется прямая (плоскость), на которой возможно параллельное асинхронное выполнение тела цикла, причем в отличие от метода координат, эта прямая (плоскость) может иметь наклон по отношению к осям координат. Цикл вида:

```
      DO 5  I = 2,N
      DO 5  J = 2,M
    5 A(I,J) = ( A(I-1,J) + A(I,J-1) ) * 0.5
```

методом координат не векторизуется. Действительно, при фиксированном значении переменной I (I = i) значение, вычисленное в точке (i,j) пространства итераций, зависит от результата вычислений в предыдущей точке (i,j-1) , так что параллельное выполнение тела цикла по переменной J невозможно. Аналогично нельзя проводить параллельные вычисления по переменной I.

Однако можно заметить, что результат будет также правильным, если вычисления проводить в следующем порядке:

на 1-м шаге - в точке (2,2),
на 2-м шаге - в точках (3,2) и (2,3),
на 3-м шаге - в точках (4,2), (3,3) и (2,4),
на 4-м шаге - в точках (5,2), (4,3), (3,4) и (2,5)

Вычисления в указанных точках для каждого шага, начиная со второго, можно проводить параллельно и асинхронно. Перечисленные кортежи точек лежат на параллельных прямых вида I+J=K , а именно: на первом шаге - на прямой I+J=4 , на втором - I+J=5, на третьем шаге - I+J=6 и т.д., на последнем ((N-2)+(M-2)+1) - ом шаге - на прямой I+J=M+N.

В общем случае для n-мерного тесногнездового цикла ищется семейство параллельных гиперплоскостей в n-мерном пространстве итераций, таких что во всех точках каждой из этих гиперплоскостей возможно параллельное выполнение тела цикла.

Для приведенного примера множество точек, в которых возможно параллельное выполнение, является однопараметрическим (прямая) и определяется из решения уравнения 1I+J=K 0. Цикл (5) может быть преобразован к виду:

```
DO 5   K = 4,M+N
Тн = MMAX(2,K-N)
Тк = MMIN(M,K-2)
DO 5  T = Тн,Тк  1: PAR
I  = T
J  = K - T
5  A(I,J) = ( A(I-1,J) + A(I,J-1) ) * 0.5
```

Функция MMAX(X,Y) выбирает ближайшее целое, большее или равное максимуму из чисел X и Y , а функция MMIN(X,Y) - ближайшее целое, меньшее или равное минимуму из X и Y .

Внутренний цикл по переменной T может выполняться параллельно для всех значений T . Границы изменения переменной цикла T меняются при переходе с одной прямой (гиперплоскости) на другую, поэтому их необходимо перевычислять во внешнем цикле. Число итераций внутреннего цикла, то есть потенциальная длина векторной операции, меняется с изменением K . Для приведенного примера диапазон изменения T сначала возрастает, а потом убывает, причем для начального и конечного значения K он равен единице. В некоторых случаях для отдельных значений K накладные расходы на организацию векторного вычисления могут превысить эффект ускорения от векторного выполнения.

Вопрос оценки целесообразности проведения векторизации данным методом должен рассматриваться для каждого конкретного случая отдельно.

66. Метод параллелепипедов.

Идея метода заключается в выявлении зависимых итераций цикла и объединении их в последовательности - ветви, которые могут быть выполнены независимо друг от друга. П ри этом в пространстве итераций определяются области (параллелепипеды), все точки которых принадлежат разным ветвям. Задача максимального распараллеливания заключается в поиске параллелепипеда наибольшего объема; тогда исходный цикл выполняется наибольшим числом параллельных ветвей, каждая из которых представляет собой исходный цикл, но с другим шагом изменения индекса.
Для исходного цикла:

```
      DO 7  I = 1,7
      DO 7  J = 1,3
    7 X(I,J)  = X(I-2,J-1)
```

параллельное представление в виде:

```
      DO 7  (K,L) = (1,1) (P1,P2)   1:  PAR
      DO 7  I = K,7,P1
      DO 7  J = L,3,P2
    7 X(I,J)  = X(I-2,J-1)
```

допускается для различных разбиений пространства итераций: пара (P1,P2) может иметь, например, значения (2,1), (2,3) или (7,1). Таким образом, исходный цикл (7) преобразуется в последовательность параллельных ветвей, имеющих циклический вид.

В общем виде задача, рассматриваемая методом параллелепипедов, для одномерных циклов состоит в определении возможности представления цикла:

```
DO L  I = 1,r
L  T(I)
```

(где T(i) - тело цикла) в виде следующей языковой конструкции:

```
DO L  K = 1,p   1: PAR
DO L  I = K,r,p
L  T(I)
```

67. Оценить возможность параллельного выполнения цикла: DO i = 2,N A(i) = (B(i) + (i))/A(i+CONST) ENDDO

Если const = 0, то все итерации цикла независимы и такой цикл может быть выполнен на любой многопроцессорной ЭВМ, включая Иллиак-4. (каждый виток цикла выполняется на отдельном процессоре)
Если const > 0 (пусть =1) то при параллельное выполнение цикла на ЭВМ класса МКМД без дополнительных мер по синхронизации работы процессоров невозможно. Например, пусть N=3 и два процессора вычисляют параллельно эти итерации, тогда, первый процессор вычисляет A2 по B2, C2, A3, а второй, A3 по B2, C2, A4 . При отсутствии синхронизации может случиться ситуация, при которой второй процессор завершит свою работу до начала работы первого. Тогда первый процессор для вычислений будет использовать A3, которое обновил второй процессор, что неверно, ибо здесь нужно “старое” значение A3.

Однако, этот цикл выполняется параллельно на ЭВМ ОКМД (SIMD) так как там этот цикл может быть выполнен такими командами:
1. Считать Bi в сумматоры каждого из n АЛУ.
2. Сложить Ci со своим содержимом сумматора.
3. Разделить содержимое каждого i-сумматора на Ai+1.
4. Записать содержимое i- сумматоров в Ai.
Из за того, что выборка из памяти и запись в память производится синхронно (одновременно), то работа цикла – корректна.
Если const < 0 то параллельное выполнение цикла невозможно, ибо для выполнения очередной итерации цикла необходимы результаты работы предыдущей (рекурсия). Однако известны приемы преобразования такого рода циклов к виду, допускающие параллельное выполнение.

68. Стандарты OpenMP.

Интерфейс OpenMP задуман как стандарт для программирования на масштабируемых SMP-системах (SSMP,ccNUMA, etc.) в модели общей памяти (shared memory model). В стандарт OpenMP входят спецификации набора директив компилятора, процедур и переменных среды.

До появления OpenMP не было подходящего стандарта для эффективного программирования на SMP-системах. Наиболее гибким, переносимым и общепринятым интерфейсом параллельного программирования является MPI (интерфейс передачи сообщений). Однако модель передачи сообщений 1) недостаточно эффективна на SMP-системах; 2) относительно сложна в освоении, так как требует мышления в "невычислительных" терминах.

Проект стандарта **X3H5** провалился, так как был предложен во время всеобщего интереса к MPP-системам, а также из-за того, что в нем поддерживается только параллелизм на уровне циклов. OpenMP развивает многие идеи X3H5. POSIX-интерфейс для организации нитей (**Pthreads**) поддерживается широко (практически на всех UNIX-системах), однако по многим причинам не подходит для практического параллельного программирования:

- нет поддержки Fortran-а,
- слишком низкий уровень,
- нет поддержки параллелизма по данным,
- механизм нитей изначально разрабатывался не для целей организации параллелизма.

OpenMP можно рассматривать как высокоуровневую надстройку над Pthreads (или аналогичными библиотеками нитей).
Многие поставщики SMP-архитектур (Sun,HP,SGI) в своих компиляторах поддерживают **спецдирективы для распараллеливания** циклов. Однако эти наборы директив, как правило, 1) весьма ограничены; 2) несовместимы между собой; в результате чего разработчикам приходится распараллеливать приложение отдельно для каждой платформы. OpenMP является во многом обобщением и расширением упомянутых наборов директив.

Директивы OpenMP с точки зрения Фортрана являются комментариями и начинаются с комбинации символов "!$OMP". Директивы можно разделить на 3 категории: определение параллельной секции, разделение работы, синхронизация. Каждая директива может иметь несколько дополнительных атрибутов - клауз. Отдельно специфицируются клаузы для назначения классов переменных, которые могут быть атрибутами различных директив.

Порождение нитей

PARALLEL ... END PARALLEL
Определяет *параллельную область* программы. При входе в эту область порождаются новые (N-1), образуется "команда" из N нитей, а порождающая нить получает номер 0 и становится основной нитью команды (т.н. "master thread"). При выходе из параллельной области основная нить дожидается завершения остальных нитей, и продолжает выполнение в одном экземпляре. Предполагается, что в SMP-системе нити будут распределены по различным процессорам (однако это, как правило, находится в ведении операционной системы).

Каким образом между порожденными нитями **распределяется работа** - определяется директивами **DO**,**SECTIONS** и **SINGLE**. Возможно также явное управление распределением работы (*а-ля MPI*) с помощью функций, возвращающих номер текущей нити и общее число нитей. По умолчанию (вне этих директив), код внутри PARALLEL исполняется всеми нитями одинаково.

Вместе с PARALLEL может использоваться клауза **IF(условие)** - й параллельная работа инициируется только при выполнении указанного в ней условия.

Параллельные области могут динамически вложенными. По умолчанию (если вложенный параллелизм не разрешен явно), внутренняя параллельная область исполняется одной нитью.

Разделение работы (work-sharing constructs)
Параллельные циклы

DO ... [ENDDO]
Определяет параллельный цикл.

Клауза **SCHEDULE** определяет способ распределения итераций по нитям:

- **STATIC**,m - статически, блоками по m итераций

- **DYNAMIC**,m - динамически, блоками по m (каждая нить берет на выполнение первый еще невзятый блок итераций)
- **GUIDED**,m - размер блока итераций уменьшается экспоненциально до величины m
- **RUNTIME** - выбирается во время выполнения .

По умолчанию, в конце цикла происходит неявная синхронизация; эту синхронизацию можно запретить с помощью **ENDDO NOWAIT**.

Параллельные секции

SECTIONS ... END SECTIONS
Не-итеративная параллельная конструкция. Определяет набор независимых секций кода (т.н., "конечный" параллелизм). Секции отделяются друг от друга директивой **SECTION**.

Примечание. Если внутри PARALLEL содержится только одна конструкция DO или только одна конструкия SECTIONS, то можно использовать укороченную запись: **PARALLEL DO** или **PARALLEL SECTIONS**.

Исполнение одной нитью

SINGLE ... END SINGLE
Определяет блок кода, который будет исполнен только одной нитью (первой, которая дойдет до этого блока).

Явное управление распределением работы

С помощью функций **OMP_GET_THREAD_NUM()** и **OMP_GET_NUM_THREADS** нить может узнать свой номер и общее число нитей, а затем выполнять свою часть работы в зависимости от своего номера (этот подход широко используется в программах на базе интерфейса MPI).

Директивы синхронизации

- **MASTER ... END MASTER**
  Определяет блок кода, который будет выполнен только master-ом (нулевой нитью).
- **CRITICAL ... END CRITICAL**
  Определяет критическую секцию, то есть блок кода, который не должен выполняться одновременно двумя или более нитями
- **BARRIER**
  Определяет точку барьерной синхронизации, в которой каждая нить дожидается всех остальных.
- **ATOMIC**
  Определяет переменную в левой части оператора "атомарного" присваивания, которая должна корректно обновляться несколькими нитями.
- **ORDERED ... END ORDERED**
  Определяет блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле. Может использоваться для упорядочения вывода от параллельных нитей.
- **FLUSH**
  Явно определяет точку, в которой реализация должна обеспечить одинаковый вид памяти для всех нитей. Неявно FLUSH присутствует в следующих директивах: BARRIER, CRITICAL, END CRITICAL, END DO, END PARALLEL, END SECTIONS, END SINGLE, ORDERED, END ORDERED.

В целях синхронизации можно также пользоваться механизмом замков (locks).

В OpenMP переменные в параллельных областях программы разделяются на два основных класса:
- SHARED (общие; под именем A все нити видят одну переменную) и
- PRIVATE (приватные; под именем A каждая нить видит свою переменную).

Отдельные правила определяют поведение переменных при входе и выходе из параллельной области или параллельного цикла: REDUCTION, FIRSTPRIVATE, LASTPRIVATE, COPYIN.

В целях создания переносимой среды запуска параллельных программ, в OpenMP определен ряд переменных среды, контролирующих поведение приложения.
В OpenMP предусмотрен также набор библиотечных процедур, которые позволяют:

- во время исполнения контролировать и запрашивать различные параметры, определяющие поведение приложения (такие как число нитей и процессоров, возможность вложенного параллелизма); процедуры назначения параметров имеют приоритет над соответствующими переменными среды.
- использовать синхронизацию на базе замков (locks).

69. Язык Фортран-DVM.

DVM-модель выполнения параллельной программы охватывает параллелизм по данным и параллелизм задачи (см. также проект [ИСП РАН/Java-DVM]). В рамках DVM-модели программист (в рамках пространства глобальных имен - единое адресное пространство) определяет:

- распределение элементов массивов по процессорам;
- распределение витков циклов по процессорам;
- организацию эффективного доступа к данным, расположенным на иных процессорах;
- организацию эффективного выполнения редукционных операций (например, sum, min, max - элементов распределенного массива).

Язык Фортран-DVM: Фортран-77, со вставкой специальных комментариев.

- Компиляция для однопроцессорных (последовательных) платформ: спецкомментарии прозрачны для стандартных компиляторов;
- Компиляция для многопроцессорных платформ: собственный Фортран-DVM компилятор расширяет спецкомментарии в конструкции с обращениями к библиотеке run-time support.

Язык Си-DVM: стандартный Си, с библиотекой специальных макросов (с "пустым" определением макросов), реализующих DVM-модель.

- Компиляция для однопроцессорных (последовательных) платформ: спецмакросы прозрачны для стандартных компиляторов, так как они имеют пустое определение;
- Компиляция для многопроцессорных платформ: собственный Си-DVM компилятор расширяет спецмакросы в конструкции с обращениями к библиотеке run-time support.

Имеются средства трассировки (последовательного и параллельного исполнения одной и той же программы), сравнения трасс, измерения параметров эффективности (распараллеливания) исполнения программ.

Компиляция программ на языках Фортран/Си-DVM поддержана на платформы:

- обычные (последовательные) ЭВМ со стандартным языком Фортран-77/Си (последовательная отладка и т.п.);
- параллельные ЭВМ с MPI + Фортран/Си (в том числе: МВС-100);
- параллельные ЭВМ с PVM + Фортран/Си (в том числе: HP Convex SPP-1000);
- параллельные ЭВМ с GNS или ОС Router + Фортран/Си (в том числе: МВС-100).

Язык Fortran DVM (FDVM) представляет собой язык Фортран 77 [5], расширенный спецификациями параллелизма. Эти спецификации оформлены в виде специальных комментариев, которые называются *директивами*. Директивы FDVM можно условно разделить на три подмножества:

- Распределение данных
- Распределение вычислений
- Спецификация удаленных данных

Модель параллелизма FDVM базируется на специальной форме параллелизма по данным: одна программа – множество потоков данных (ОПМД). В этой модели одна и та же программа выполняется на каждом процессоре, но каждый процессор выполняет свое подмножество операторов в соответствии с распределением данных.

В модели FDVM пользователь вначале определяет многомерный массив виртуальных процессоров, на секции которого будут распределяться данные и вычисления. При этом секция может варьироваться от полного массива процессоров до отдельного процессора.

На следующем этапе определяются массивы, которые должны быть распределены между процессорами (*распределенные данные*). Эти массивы специфицируются директивами отображения данных (раздел 4). Остальные переменные (распределяемые по умолчанию) отображаются по одному экземпляру на каждый

процессор (*размноженные данные*). Размноженная переменная должна иметь одно и то же значение на каждом процессоре за исключением переменных в параллельных конструкциях

Модель FDVM определяет два уровня параллелизма:

- параллелизм по данным на секции массива процессоров;
- параллелизм задач – независимые вычисления на секциях массива процессоров.

Параллелизм по данным реализуется распределением витков тесно-гнездового цикла между процессорами. При этом каждый виток такого параллельного цикла полностью выполняется на одном процессоре. Операторы вне параллельного цикла выполняются по правилу собственных вычислений.

Параллелизм задач реализуется распределением данных и независимых вычислений на секции массива процессоров .

При вычислении значения собственной переменной процессору могут потребоваться как значения собственных переменных, так и значения несобственных (*удаленных*) переменных. Все удаленные переменные должны быть указаны в директивах доступа к удаленным данным.

70. Язык Sisal.

!! вроде как - язык описания поточных алгоритмов
Sisal - функциональный язык программирования. Программист не заботится о параллельных свойствах программ, компилятор определяет все зависимости, распределяет работу по процессорам, вставляет необходимые пересылки и синхронизации.

Заметное место среди языков функционального программирования занимают языки организации распределенных и параллельных вычислений. Практики с большой похвалой отзываются о языке функционального программирования Erlang фирмы Ericsson. Здесь мы рассмотрим один из довольно известных — язык функционального программирования SISAL [11].

- создание универсального функционального языка;
- разработка техники оптимизации для высокоэффективных параллельных программ;
- достижение эффективности исполнения, сравнимой с императивными языками типа Fortran и C;
- внедрение функционального стиля программирования для больших научных программ.

Название языка расшифровывается как "Streams and Iterations in a Single Assignment Language", сам он представляет собой дальнейшее развития языка VAL, известного в середине 70-х годов. Среди целей разработки языка SISAL следует отметить наиболее характерные, связанные с функциональным стилем программирования:

Эти цели создателей языка SISAL подтверждают, что функциональные языки способствуют разработке корректных параллельных программ. Одна из причин заключается в том, что функциональные программы свободны от побочных эффектов и ошибок, зависящих от реального времени. Это существенно снижает сложность отладки. Результаты переносимы на разные архитектуры, операционные системы или инструментальное окружение. В отличие от императивных языков, функциональные языки уменьшают нагрузку на кодирование, в них проще анализировать информационные потоки и схемы управления. Легко создать функциональную программу, которая является безусловно параллельной, если ее можно писать, освободившись от большинства сложностей параллельного программирования, связанных с выражением частичных отношений порядка между отдельными операциями уровня аппаратуры. Пользователь Sisal-а получает возможность сконцентрироваться на конструировании алгоритмов и раз работке программ в терминах крупноблочных и регулярно организованных построений, опираясь на естественный параллелизм уровня постановки задачи.

Начнем с примера программы:

1. Вычисление числа π (пи).

```
For     % инициирование цикла
   Approx := 1.0;
   Sign := 1.0;
   Denom := 1.0;
   i := 1

while i <= Cycles do   % предусловие завершения цикла
   Sign := -Sign;                    % однократные
   Denom := Denom + 2.0;             % присваивания
```

```
   Approx := Approx + Sign / Denom;    % образуют
   i := i + 1                          % тело цикла

returns Approx * 4.0
% выбор и вычисление результата цикла
end for
```

2. Это выражение также вычисляет число π (пи).

```
for i in [1..Cycles/2] do
               % пространство параллельно
               % исполнимых итераций

   val := 1.0/real(4*i-3) — 1.0/real(4*i-1);
               % тело цикла, для каждого i
               % исполняемое независимо

returns sum( val )   % выбор и свертка результатов
               % всех итераций цикла

end for * 4.0   % вычисление результата
               % выражения
```

Это выражение вычисляет сумму всех вычисленных значений val и умножает результат на 4.0.

3, 4. В for-выражениях операции dot и cross могут порождать пары индексов при формировании пространства итерирования:

```
for i in [1..2] dot j in [3..4] do
               % для пар индексов [1,3] и
               % [2,4]
   returns product (i+j)
               % произведение сумм
   end for      % = 24
for i in [1..2] cross j in [3..4] do
               % для пар [1,3], [1,4], [2,3]
               % и [2,4]
   returns product (i+j)
               % произведение сумм
   end for      % = 600
```

5. Итеративное for-выражение с обменом данными между итерациями:

```
for
   I := 1
while I < S do
   K := I;
   I := old I + 2;
   % значение из предыдущей итерации
   J := K + I;
returns product(I+J)
end for
```

Как это свойственно языкам функционального программирования, Sisal язык математически правильный — функции отображают аргументы в результаты без побочных эффектов, и программа строится как выражение, вырабатывающее значение. Наиболее интересна форма параллельного цикла. Она включает в себя три части: генератор пространства итераций, тело цикла и формирователь возвращаемых значений.

SISAL-программа представляет собой набор функций, допускающих частичное применение, т.е. вычисление при неполном наборе аргументов. В таком случае по исходному определению функции строятся его проекции, зависящие от остальных аргументов, что позволяет оперативно использовать эффекты смешанных вычислений и определять специальные оптимизации программ, связанные с разнообразием используемых конструкций и реализационных вариантов параллельных вычислений.

```
function Sum (N);   % Сумма квадратов
   result (+ ( sqw (1 .. N)));
```

Обычно рассматривают оптимизации, обеспечивающие устранение неиспользуемого кода, чистку циклов, слияние общих подвыражений, перенос участков повторяемости для обеспечения однородности распараллеливаемых ветвей, раскрутку или разбиение цикла, втягивание константных вычислений, уменьшение силы операций, удаление копий агрегатных конструкций и др.

71. Система программирования Норма.

Декларативный язык Норма предназначен для описания решения вычислительных задач сеточными методами. Возможно применение для иных задач (нет более подробных данных).

Высокий уровень абстракции языка позволяет описывать задачи в нотации, близкой к исходной постановки проблемы математиком (программирование без программиста), получать описание не ориентированное на конкретную архитектуру и/или конкретные методы организации параллельного выполнения. Язык не содержит традиционные конструкции языков программирования, фиксирующие порядок вычисления и/или иным образом "скрывающие/ограничивающие" параллелизм (например, COMMON-блоки).

После двух фаз компиляции (анализ информационных зависимостей и генерация ярусно-параллельного графа алгоритма) вывод результирующей программы возможен в следующих форматах:

- Фортран 77-для выполнения на последовательных ЭВМ;
- Фортран/PVM-для выполнения на любых системах с Фортран/PVM;
- Фортран/GNS-для выполнения на МВС-100 и других системах с Фортран/GNS-см. проект ИПМ/GNS;
- Фортран/Convex-для выполнения на HP Convex SPP-100 и других системах с Фортран/Convex;
- (в разработке) Фортран/MPI-для выполнения на любых системах с Фортран/MPI;

http://parallel.ru/tech/norma/

Непроцедурный язык Норма предназначен для записи численных методов решения задач математической физики разностными методами. Процесс разработки программ для решения таких задач можно разбить на следующие этапы:

- Постановка задачи. Выходом этого этапа является обычно система дифференциальных уравнений, описывающих задачу.
- Выбирается пространственно-временная сетка и производится дискретизация уравнений с помощью одного из разностных методов.
- Производится выбор метода решения дискретных уравнений. В результате получаются формулы (соотношения), описывающие необходимые вычисления в узлах сетки.
- Полученные формулы программируются на некотором языке, который обеспечивает решение задачи на вычислительной машине.

Главная идея, положенная в основу языка Норма, заключается в том, что полученное на третьем этапе описание решения задачи, почти непосредственно используется для ввода его в вычислительную систему и проведения счета.

Таким образом, язык Норма дает прикладному математику возможность сформулировать свою задачу в привычных для него терминах. Организация процесса вычислений с учетом архитектуры ЭВМ (возможностей параллельной, векторной обработки и т.п.) - это задача транслятора с языка Норма.

Запись на языке Норма - это, по существу, строгая запись численных методов решения математической задачи, запись еще не алгоритмов, а просто расчетных формул и остальной необходимой информации, которую необходимо знать, чтобы написать программу для ЭВМ.

Расчетные формулы, получаемые прикладным специалистом, обычно записываются в виде соотношений. Отметим, что в записи на Норме не требуется никакой информации о порядке счета, способах организации вычислительных (циклических) процессов. Порядок предложений языка может быть произвольным - информационные взаимосвязи будут выявлены и учтены при организации процесса счета транслятором.

Выбор уровня языка Норма определяет характерную его черту - в этом языке нет необходимости вводить такие понятия, как оператор присваивания и возможность переприсваивания значений (типа X:=X+1) и операторы перехода. Наличие таких понятий в традиционных языках программирования объясняется необходимостью

формулировки конкретного алгоритма с учетом вопросов экономии и распределения памяти, порядка выполнения операторов и т.п. Норма - это язык с однократным присваиванием Побочный эффект в языке Норма отсутствует по определению.

Понятно, что многие из этих вопросов появляются снова на этапе синтеза рабочей программы. Однако, здесь они решаются автоматически по строгим правилам, гарантирующим правильность синтезируемой программы.

Высокий уровень языка обеспечивает дружественный интерфейс с пользователем, причем даже ошибки, которые обнаруживаются транслятором-синтезатором, также фиксируются в терминах предметной области. Автоматический синтез целевой программы по исходной Норма-программе гарантирует правильность целевой программы (с точностью до правильности работы транслятора-синтезатора).

Запись на таком языке может помещаться в библиотеку исходных описаний решения задачи. Если при этом имеется описание непрерывного уравнения и указывается метод дискретизации, то текст Норма-программы достаточно легко понять, так как он содержит только математические (долго живущие) понятия.

Важно отметить, что в записи на Норме отсутствуют избыточные связи, которые обычно накладываются при программировании, особенно при оптимизации алгоритмов. Эти связи часто ограничивают возможности распараллеливания. Например, конструкция ***COMMON*** языка Фортран обычно ограничивает автоматическое распараллеливание программ.

Не менее важной, а может быть и наиболее важной с точки зрения обеспечения дружественного интерфейса с пользователем, является возможность использования языка Норма в качестве базиса для создания интегрированной среды разработки прикладных программ.

Компонентами такой среды могут быть диалоговые средства, средства визуализации, средства отладки в содержательных терминах, синтаксически-ориентированный редактор, графический редактор и так далее. Этот перечень можно считать более или менее стандартным "джентльменским набором" подсистем, составляющих современную среду разработки.

Кроме этого, язык Норма может оказаться необходимым промежуточным уровнем представления информации при сквозной автоматизации процесса решения прикладной задачи от разработки метода решения до проведения расчетов.

72. Распараллеливание алгоритмов сложения методом редукции

Параллельно суммирование последовательности n чисел можно произвести так: на первом этапе складываются соседние числа. Полученные суммы также складываются попарно, и т.д. Для n = 2**q алгоритм состоит из q = log n этапов, на первом этапе выполняются n/2 сложений (степень параллельности этапа n/2), на втором - n/4 и т. д. Такой алгоритм называется сложение методом сдваивания, он имеет различную степень параллелизма на разных этапах вычислений. Граф, описывающий последовательность операций сложения, граф сдваивания (по Д. Ортега "fan-in grafh.") представляет собой двоичное дерево, соответственно, выполняемые операции можно называть операциями на дереве.

Способ реализации процедуры суммирования данным методом зависит от архитектуры вычислительной системы. При наличии n/2 процессоров эту работу можно выполнить так: на первом этапе одновременно получить суммы четных/нечетных соседних элементов последовательности Ai, т.е. (A1+A2), (A3+A4),...(An-1+An); затем такая процедура повторяется для суммирования полученных частных сумм и так далее. Если n = 2**q, то через q = log2n шагов получается искомая сумма. Однако, потери на синхронизацию вычислений, на пересылки частных сумм могут оказаться сравнимы с временем вычисления суммы двух чисел в каждом процессоре.

Поэтому, с учетом особенностей характеристик вычислительной системы, дерево вычислений может быть преобразовано, например, с целью увеличения числа операций, выполняемых в узлах, повышения "зернистости" алгоритма.

Алгоритм сдваивания реализуются также в блоках оптимизации компиляторов последовательных ЭВМ для полной загрузки конвейерных вычислителей. Так алгоритм оптимизация "балансировка дерева вычислений" (tree-height reduction or balancing) будет трактовать вычисление суммы вещественных чисел: A+B+C+D+E+F+G+H, как последовательность операций: (((A+B)+(C+D))+((E+F)+(G+H))).

Рекурсия - последовательность вычислений, при котором значение самого последнего терма в последовательности зависит от одного или несколько ранее вычисленных термов. Пусть группа вычислений может производиться параллельно, использую результаты вычислений, выполненных на предыдущих этапах (полученных в виде начальных данных). Тогда, каждая группа вычислений называется "ярусом" параллельной формы, число групп - "высотой", максимальное число операций в группе "шириной" параллельной формы. Один и тот же алгоритм

может иметь несколько представлений в виде параллельных форм, различающиеся как шириной, так и высотой. Редукционный алгоритм сдваивания для суммирования чисел с получением частных сумм может иметь вид:

```
Данные     A1    A2    A3    A4    A5    A6    A7    A8
Ярус 1     A1+A2       A3+A4       A5+A6       A7+A8
Ярус 2    A12+A3     A12+A34     A56+A7      A56+A78
Ярус 3  A1234+A5   A1234+A56   A1234+A567  A1234+A5678
```

Высота параллельной формы равна трем, ширина - четырем, причем загрузка вычислителей (четырех) полная.

В данном алгоритме производится вычисления пяти 'лишних' чисел по сравнению с последовательным алгоритмом.

Каскадное суммирование
Примером параллельных алгоритмов, ориентированных на векторные вычислители, может служить метод вычисления каскадных сумм (алгоритм рекурсивного удвоения) для распараллеливания операций суммирования. Пусть необходимо просуммировать n чисел с сохранением промежуточных сумм: Si = Si-1 + Ai i = 2,..n, S1 = A1. Исходный вектор A поэлементно складывается с вектором As, полученный из исходного со сдвигом на один элемент и заполнением позиции элемента A0 нулем. Для вектора результата процедура повторяется, но сдвиг - на 2 позиции. Если n = 2**k, то через k операций получается вектор результата.
Для i=8:

```
A1    0     A1      0      A1       0         A1
A2    A1    A12     0      A12      0         A12
A3    A2    A23     A1     A123     0         A123
A4 +  A3 =  A34  +  A12 =  A1234 +  0      =  A1234
A5    A4    A45     A23    A2345    A1        A12345
A6    A5    A56     A34    A3456    A12       A123456
A7    A6    A67     A45    A4567    A123      A1234567
A8    A7    A78     A56    A5678    A1234     A12345678
```

Алгебра данного метода может быть записана в виде вычисления (возможно, параллельного) частных сумм вида: Si = Ali, где Ali = A(l-1)i + A(l-1)(i-2**(l-1)), A0i = Ai для i = 1,2,...n.
Вычисления проводятся l = 0,1,...,log2n раз, причем, если у Ali индекс i выходит из интервала 1<= i <= n то он принимается равным нулю.
Хокни предлагает элегантную векторную форму записи алгоритма каскадного суммирования массива D(n):

```
X = D
DO L = 1,LOG2(N)
   X = X + SHIFTR(X,2**(L-1))
ENDDO
```

Результат векторной функции SHIFTR(A,l) есть массив (вектор), полученный из A , элементы которого сдвинуты на L позиций вправо, а L левых элементов заполнены нулями. Практическая реализация алгоритма может исключить излишние операции сложения с нулем, однако, и после этого, по сравнению с последовательным алгоритмом, данный - требует лишние операции.

73. Метод распараллеливания алгоритма общей рекурсии 1-го порядка.

Редукция - упрощение, в биологии уменьшение размера органа вплоть до его полного исчезновения. Циклическая редукция - алгоритмы численного анализа для распараллеливания последовательных алгоритмов, основанный на последовательном, циклическом применении параллельных вычислений, число которых на каждом этапе уменьшается (делится) пополам.

Линейной рекурсией 1 порядка называется система уравнений вида:

```
X1 = D1
X2 = X1 * A2 + D2
.................
Xi = Xi-1 * Ai + Di
.................
Xn = Xn-1 * An + Dn
```

в общем виде: Xi = Xi-1 * Ai + Di, i = 2,3,...n, X1 = D1
Это система эквивалентна двухдиагональной системе уравнений Ax=d, где

```
        ⎡    1          ⎤          ⎡  d1  ⎤
  A =   ⎢ -a2 1           ⎥    d =   ⎢  .   ⎥
        ⎢   .......       ⎥          ⎢      ⎥
        ⎣      -an 1      ⎦          ⎣  dn  ⎦
```

Последовательный алгоритм вычислений может быть записан так:

```
X(1) = A(1) + D(1)
DO i = 1,n
      X(i) = X(i-1) * A(i) + D(i)
ENDDO
```

Рекурсивная зависимость итераций цикла не позволяет ускорить вычисления за счет параллельной работы оборудования. Преобразуем данный алгоритм в параллельный методом циклической редукции. Рассмотрим два соседних уравнения:

```
Xi-1 = Xi-2 * Ai-1 + Di-1
Xi = Xi-1 * Ai + Di
```

и подставив первое во второе, получаем:

```
Xi = (Xi-2 * Ai-1 + Di-1) *  Ai + Di = Xi-2 * A1i + D1i , где
A1i = Ai * Ai-1 ,
D1i = Ai * Di-1 + Di
```

Тогда, проведя эту операцию для всей системы уравнений, получим систему уравнений порядка n/2. Если повторить процедуру l раз (если n = 2**l), то в результате получается значение: Xn = Dnl. Для получения полного вектора X необходимо модифицировать алгоритм, например, по аналогии с алгоритмами суммирования.

Очевидно, что вычисления Aji и Dji можно проводить параллельно методом каскадных сумм с сохранением частных сумм. Приведенные уравнения для уровня i имеют вид:

```
Xi = Ali * Xi-2**l + Dli , где l = 0,1,..,log2n , i = 1,2,..,n
Ali = Al-1i * Al-1(i-2**l-1)
Dli = Al-1i * Dl-1(i-2**l-1) + Dl-1i
Начальные данные: A0i = Ai, D0i = Di
```

Если индекс i у любого Ali, Dli и Xi попадает вне диапазона 1 <= i <= n , то он должен быть приравнен к нулю. Тогда , при l = log2n в уравнениях: Xi = Ali * Xi-2**l + Dli индекс Xi-2**l = Xi-n находится вне диапазона, и, следовательно, решением системы уравнений будет вектор: Xi = Dli

Нотация Хокни для данного алгоритма:

```
X = D
DO L = 1,LOG2(N)
      X = A * SHIFTR(X,2**(L-1)) + X
      A = A * SHIFTR(A,2**(L-1))
ENDDO
```

74. Представление машинных чисел.

Подмножество вещественных чисел, которое может быть представлено в ЭВМ в форме чисел с плавающей запятой, принято обозначать буквой F и определять его элементы для конкретной архитектуры - "машинные числа", (по Форсайту и др.) четырьмя целочисленными параметрами: базой b, точностью t и интервалом значений показателя [L,U]. Множество F содержит число нуль и все f числа вида: f = (+/-).d1d2...dt * b**e, где e называется показателем, число .d1d2...dt = (d1/b+ ....+dt/(b**t)) - дробной частью - мантиссой, причем: 0<=di<b, L<=e<=U. Каноническая или нормализованная форма F определяется дополнительным соотношением d1 =/= 0; это условие позволяет устранить неоднозначность представления одинаковых чисел, дает наивысшую возможную точность представления чисел. Особенности F:

- для каждого ненулевого f верно: m<=|f|<=M, где m = b**(L-1), M = (b**U) * (1-b**(-t));
- множество F конечно и содержит 2*(b-1)*(b**(t-1))*(U-L+1)+1 чисел, которые отстоят друг от друга на числовой оси на неравные промежутки.

75. Арифметика машинных чисел.

76. Погрешности при вычислениях чисел на параллельных системах. Оценить полную ошибку суммирования положительных чисел.

Формулы оценки абсолютной и относительной погрешности арифметических операций.

1. Сложение: X=X1+X2 X1>0 , X2>0

Абсолютная погрешность: Dx = Dx1 + Dx2 , относительная: dx = dx1 + dx2

2. Вычитание: X=X1-X2 X1>X2>0

Абсолютная погрешность:Dx=Dx1 + Dx2, относительная: dx=(X1*dx1 + X2*dx2)/X

Если X1 >> (много больше) X2, то dx (почти равно) dx1.

Если X1 (почти равно) X2, то dx будет очень велико. При вычитании близких по величине чисел получается большая потеря верных знаков.

3. Произведение: X = X1+X2

Погрешности: Dx = (X/X1)*Dx1 + (X/X2)*Dx2 dx = dx1 + dx2

4. Частное двух величин: X = X1/X2

Погрешности:Dx = (|X2|*Dx1 + |X1|*Dx2)/X2**2 , dx = dx1 + dx2

Формулы дают выражения для определения ошибки результата каждого из 4 арифметических действий как функции от величины чисел, участвующих в операциях, и абсолютных ошибок для них (например, погрешностей исходных данных). Для определения полной ошибки результата нужно к этим ошибкам отдельно добавить ошибки округления.

Пример расчета полной ошибки для суммирования положительных чисел (Г.К. Боровин).

Формула полной ошибки для суммирования положительных чисел Ai(i=1,..,n) имеет вид:

Ds = A1*da1 + A2*da2 +...+ An*dan + d1*(A1+A2) +..+ d(n-1)*(A1+..+An) + dn , где

dai - относительные ошибки представления чисел в ЭВМ, a di - относительные ошибки округления чисел при каждой следующей операции сложения.

Пусть: все dai = da и di = d , a Ks = A1+A2+..+An, тогда:

Ds = da*Ks + d*[(n-1)*A1+(n-1)*A2 +...+ 2*A(n-1) + An]

Очевидно, что наибольший "вклад" в сумму ошибок вносят числа, суммируемые вначале. Следовательно, если суммируемые положительные числа упорядочить по возрастанию, максимально возможная ошибка суммы будет минимальной. Изменяя порядок суммирования чисел можно получать различные результаты. Но если даже слагаемые отличаются друг от друга незначительно, на точность результата может оказать влияние способ суммирования. Пусть суммируются 15 положительных чисел, тогда ошибка результата: Ds = da*Ks + d*(14*A1+14*A2+13*A3+....+2*A14+A15).

Слагаемое da*Ks не зависит от способа суммирования, и далее не учитывается.

Пусть слагаемые имеют вид: Ai = A0+ei, где i=1,...,15, тогда: Dss = 199*(A0+em)*d, где em = max(ei), d - ошибка округления при выполнении арифметической операции сложения.

Если провести суммирование этих чисел по группам (три группы по четыре числа и одна группа из трех чисел), то ошибки частных сумм имеют вид:

Ds1 = d*(3*A1+3*A2+2*A3+A4) <= 9*d*(A0+em)

Ds2 = d*(3*A5+3*A6+2*A7+A8) <= 9*d*(A0+em)

Ds3 = d*(3*A9+3*A10+2*A11+A12) <= 9*d*(A0+em)

Ds4 = d*(3*A13+2*A14+A15) <= 5*d*(A0+em)

а полная оценка ошибок округления будет Ds <= 32*d*(A0+em), что меньше Dss. Итак суммирование по группам дает меньшую ошибку результата.

Например, разделив процесс суммирования массива положительных чисел на параллельные процессы, и затем получив сумму частных сумм, можно получить результат, отличный от последовательного суммирования в одном процесс.

77. Точность плавающей арифметики. Машинный эпсилон.

Точность плавающей арифметики можно характеризовать посредством машинного эпсилона.
Максимальное число E такое, что 1.+ E = 1. является мерой точности представления чисел на данной ЭВМ (машинное эпсилон). Грубая схема вычисления эпсилона:

```
      EPS = 1.0
  1   EPS = 0.5 * EPS
      EPS1 = EPS + 1.0
      IF (EPS1 .GT. 1.0) GO TO 1    >
```

Погрешности округления при вычислениях
При выполнении арифметических операций на ЭВМ могут появляться числа с большим количеством значащих цифр, чем у исходных чисел, и большим, чем может быть представимо на данной ЭВМ. Например, при умножении, число значащих цифр может удвоиться. Поэтому необходимо проводить округление результата вычислений.

Простейший способ округления состоит в отбрасывания младших разрядов. Пусть при вычислении получен неотрицательный результат (q-ичная дробь): `X = 0.An A(n-1) A(n-2)....As A(s-1) A(s-2)....A1 .`

Тогда, округленное число есть: Xs= 0.An A(n-1)A(n-2)....As , а ошибка округления (абсолютная): `Dx=|Xs-X|<=q**(-[n-(s-1)]),` где n-(s-1) число значащих цифр в Xs. Максимально возможная относительная ошибка при этом будет равна:
`dx = Dx/X <= (q**(-[n-(s-1)]))/(1/q) = q**(-(n-s)).`

Другим общепринятым способом округления считается "симметричное" округление, при котором:
`Xs = Xs + q**(-(n-s)), если |Xs-X| >= 0.5**q(-(n-s)) и`
`Xs = Xs,  если |Xs-X| <  0.5**q(-(n-s))`
При этом способе округления максимально возможное значение относительной ошибки: dx <= 0.5*q**(-(n-s)+1).

78. Перечислить алгоритмы оптимизации объектных программ, которые могут повлиять на точность вычислений.

Оптимизационные преобразования программ для их оптимального выполнения на конвейерных вычислителях могут проводиться системами программирования. Эти преобразования, алгебраически эквивалентные, могут нарушить порядок вычислений, предписанный исходным текстом программы.
Последствия таких преобразований обсуждались выше. Наиболее характерные преобразования следующие.

1. Балансировка дерева вычислений

Балансировка дерева вычислений (tree-height reduction or balancing) выражений позволяют использовать конвейерное АУ без пропуска рабочих тактов. Так, вычисление суммы вещественных чисел: A+B+C+D+E+F+G+H, будет запрограммировано как последовательность операций: (((A+B)+(C+D))+((E+F)+(G+H))); это нарушает заданную по умолчанию последовательность вычислений с накоплением одной частной суммы и может повлиять на результат.

2. Исключение общих подвыражений

Алгоритмы исключения общих подвыражений (Common subexpession elimination) также могут изменить порядок вычислений.
Если компилятор распознает в выражениях повторяющееся вычисление, то это вычисление производятся один раз, его результат сохраняется на регистре, и в дальнейшем используется этот регистр. Тем самым исключается избыточность вычислений.

```
      X = A + B + C + D      ---->        REG = B + C
      Y = B + E + C                       X   = A + D + REG
                                        Y   = E + REG
```

3. Разворачивание циклов

Разворачивание циклов (loop unrolling) - расписывание цикла последовательностью операторов присваивания: либо полностью, либо размножение тела цикла с некоторым коэффициентом (фактором) размножения.
Производится частичное или полное разворачивание цикла в последовательный участок кода. При частичном разворачивании используется так называемый фактор разворачивания (который можно задавать в директиве компилятору).

```
DO I=1,100                    DO I=1,100,4
  A(I) = B(I) + C(I)            A(I) = B(I) + C(I)
ENDDO                           A(I+1) = B(I+1) + C(I+1)
                                A(I+2) = B(I+2) + C(I+2)
                                A(I+3) = B(I+3) + C(I+3)
                              ENDDO
```

При этом преобразовании снижается количество анализов итерационной переменной. Данный алгоритм также может привести к нарушению предписанного первоначально порядка вычислений. Например:

```
DO I=1,10          DO I=1,10,2
S = S + A(I)         S = S + A(I)
ENDDO               S1 = S1 + A(A+1)
                   ENDDO
                   S = S + S1
```

Здесь, суммирование проводится отдельно для четных и нечетных элементов с последующем сложением частных сумм.

По аналогии с этим принято считать, что *парадигма в программировании* -- способ концептуализации, который определяет, как следует проводить вычисления, и как работа, выполняемая компьютером, должна быть структурирована и организована.

§9. Средства автоматического распараллеливания программ

Средства автоматического распараллеливания – наиболее быстрый способ получить параллельную программу из последовательной, но степень параллелизма кодов, полученных автоматически, ниже степени параллелизма кодов программ, в которых параллелизм закладывается программистом. Так или иначе, но машина предпочтет не распараллеливать любой подозрительный фрагмент программы, в то время как программист знает, какая часть алгоритма, не являющаяся заведомо параллельной, тем не менее может быть распараллелена.

Некоторые средства автоматического распараллеливания представлены в табл. 4.

Таблица 4

Название	**API**	**Дополнительные сведения**
BERT 77	PVM, MPI	Распараллеливает Fortran-программы
FORGExplorer		Распараллеливает Fortran-программы для SMP и MPP платформ
PIPS	OpenMP, MPI, PVM	Распараллеливает Fortran-программы
VAST/Parallel	OpenMP	Распараллеливает Fortran/C-программы для SMP-платформ