1. Классификация многопроцессорных вычислительных систем. Основные понятия параллельной обработки данных

1. **SISD** (Single Instruction, Single Data) – системы с одиночным потоком команд и одиночным потоком данных; к данному типу систем можно отнести обычные последовательные ЭВМ;
2. **SIMD** (Single Instruction, Multiple Data) – системы с одиночным потоком команд и множественным потоком данных; это многопроцессорные вычислительные системы, в которых в каждый момент времени может выполняться одна и та же команда для обработки нескольких информационных элементов; подобной архитектурой обладают, например, многопроцессорные системы с единым устройством управления;
3. **MISD** (Multiple Instruction, Single Data) – системы с множественным потоком команд и одиночным потоком данных. Относительно данного типа систем нет единого мнения – ряд специалистов говорят, что примеров конкретных ЭВМ, соответствующих данному типу вычислительных систем, не существует, и введение подобного класса предпринимается для полноты системы классификации; другие же относят к данному типу, например, системы с конвейерной обработкой данных;
4. **MIMD** (Multiple Instruction, Multiple Data) – системы с множественным потоком команд и множественным потоком данных; к подобному классу систем относится большинство параллельных многопроцессорных вычислительных систем.

Существующие параллельные ВС класса MIMD образуют два технических подкласса:

Мультипроцессорные системы (многопроцессорные)

Многопроцессорные системы с общей разделяемой памятью. Такие системы принято называть симметричными многопроцессорными системами. SMP. Основной чертой является то, что каждый процессор имеет прямой и равноправный доступ к любой точке общей памяти. К памяти процессоры подключаются через общую системную шину, при этом упрощается межпроцессорное взаимодействии, упрощается программирование, так как программа работает в едином адресном пространстве.

- Такая архитектура не пригодна для создания масштабных систем из-за большого числа конфликтов при обращении к общей памяти.
- Во всех современных машинах иерархическая память. В связи с этим возникает проблема синхронизации информации, расположенной в отдельных элементах кэш памяти и основной памяти системы. Современные системы состоят из однородных микропроцессоров с массивом общей памяти. Подключается при помощи шины или коммутатора

Для построения масштабируемых систем используются кластерные системы или архитектура типа NUMA. Неоднородный доступ к памяти. В такой архитектуре память распределена физически, но логически общедоступна. С одной стороны, это позволяет работать всем процессорам с единым адресным пространством, с другой стороны увеличивает масштабируемость системы, сокращает вероятность конфликтов при доступе к одной и той же ячейке. Когерентность решается аппаратно, в отличии от чистой SMP система с NUM архитектурой вводится 3-х уровневая память:

- Кэш память процессора
- Локальная оперативная память
- Удаленная оперативная память.

Время обращение к элементам памяти разного уровня отличается на порядки, что усложняет написание эффективных параллельных программ. В любой момент времени модули, соединенные с помощью высокоскоростного коммутатора, могут получать доступ к произвольному элементу удаленной памяти. Масштабируемость NUMa систем ограничивается размером адресного пространства, аппаратуры для поддержки когерентности, и аппаратуры для управления числа процессоров. В настоящее время максимально можно поставить 256 процессоров. Общая производительность около 200 млрд оп\сек.

У любой системы с разделяемой памятью цена растет очень быстро по сравнению с ростом производительности.

Системы с распределенной памятью. MPP

Такие системы представляют собой много процессорные системы с распределенной памяти, которые с помощью коммутационной среды объединяются в выч узлы. Каждый узел состоит из одного или нескольких процессорных элементов, собственной ОЗУ, коммутационного оборудования и подсистемы ввода вывода. Каждый узел обладает всеми компонентами для независимого функционирования. На каждом узле может функционировать либо полноценная ОС, либо ее урезанный вариант, поддерживающие базовые функции ядра.

Процессоры имеют прямой доступ только к своей локальной памяти. Доступ к памяти других узлов реализуется с помощью механизма передачи сообщений. Такая архитектура позволяет устранить конфликты при обращении к памяти и проблему когерентности кэшей. Считается, что имеется возможность наращивания числа процессоров с увеличением производительности.

Основной плюс – высокая масштабируемость.

Система с распределенной памятью подходит для выполнения параллельно независимых программ с малой интенсивностью передачи информации.

Для решения проблем, возникающих в выше представленных архитектурах. Решаются векторные системы. В их состав включаются векторные конвейерные системы. Они имеют векторно-конвейерные процессоры. Несколько таких процессоров работают одновременно над общей памятью в рамках многопроцессорных конфигураций. PVP системы являются гибридными системами.

Основной недостаток – цена.

Исследования:

- Исследования в области теории параллельных алгоритмов
- Исследования в области формальных языков методов и технологии параллельного программирования и оптимизации параллельных программ
- Исследования в области теории управления параллельными вычислительными процессами

Эффективность решения конкретной задачи является особенностью реализации параллельности процесса. В целом процесс параллельного решения задачи можно определить, как одновременное выполнение отдельных фрагментов задачи, распределенном в вычислительной задаче. Для организации таких вычислений решаемая задача должна обладать свойствами параллелизма.

- Основные понятия, связанные с параллельной обработкой данных:
- Пути достижения параллелизма:
- Независимость функционирования отдельных устройств компьютеров
- Избыточность элементов вычислительной системы (дублирование, резервирование)
- Конвейерная реализация обрабатывающих устройств

Сдерживающие факторы:

- Высокая стоимость параллельных вычислительных систем
- Необходимость обобщения последовательных программ
- Потери производительности при организации параллелизма
- Постоянное совершенствование последовательных компьютеров
- Зависимость эффективности параллелизма от учета архитектуры параллельных систем

Режимы выполнения независимых частей программы:

- Многозадачный режим (разделение времени). Используется единственные процессор, такой режим псевдопараллельный, так как активный только один процесс, все остальные находятся в очереди и ожидают доступности ресурсов. Использование режима разделения времени позволяет повысить эффективность организации вычислений за счет разделения во времени вычислительных операций и операций связанными с вводом и выводом.
- Параллельные выполнение. Предполагает, что в один момент времени может выполнятся несколько команд обработки данных, такой режим обеспечивается не только при наличии

нескольких процессоров, но в случае реализации конвейерных или векторных обрабатывающих устройств.

- Распределенные вычисления. Данный термин используется для указания параллельной обработки данных для которой используется несколько обрабатывающих устройств, удаленных друг от друга, в которых передача данных по линиям связи приводит к существенным временным задержкам. Распределенные вычисления позволяют эффективно обрабатывать данные только для параллельных алгоритмов с низкой интенсивности потоков межпроцессорных передач данных. Такие задачи обычно решаются на многомашинных вычислительных комплексах, когда несколько ЭВМ связываются по локальным или глобальным информационным сетям.

Свойства параллелизма

- Задачи с естественным параллелизмом. Такие задачи предполагают наличие в процессе своего решения совокупность операции, которые могут выполнятся одновременно без дополнительной передачи информации. Процесс решения задачи с естественным параллелизмом разделяется на несколько процессов, которые выполняются независимости, при этом не требуются специфики вычислений.
- Задачи с параллелизмом множества объектов. Выполняется обработка различных или однотипных объектов по одной и той же программе. В отличии от задач с естественным параллелизмом встречаются в ситуации, когда отдельные участки вычислений должны выделятся по-разному для объектов. Основной характеристикой является ранг т.е. количество параметров по которым должна вестись обработка. В качестве ранга задачи может выступать количество разнотипных объектов в системе.
- Задачи с параллелизмом независимых ветвей. Наиболее разработанные тип параллелизма. При решении крупных задач могут быть выделены независимые части или ветви. Условия независимости ветви задачи У от ветки Х являются:
    - Между ветвями отсутствуют функциональные связи. Выход Х не является вход Y
    - Между ветвями нет связи по памяти
    - Независимость в программном отношении
    - Взаимонезависимость по управлению.

Все задачи, решаемые одновременно могут быть связаны по данным и по управлению в общей вычислительной среде. При этом задачи разделяются по степени связывания. Степень связывания определяет зернистость параллелизма. Пусть существует некоторая сложная задача, которая может быть разбита, но множество простых задач, тогда задача называется несвязанной если все элементы матрицы связности являются нулевыми. Одиночные задачи соединяются в сложную несвязную задачу в виде набора простых, вычисления при решении простых задач ведутся независимо друг от друга, обмен происходит довольно редко. Объектом распараллеливания являются простые задачи. Такие вычисления являются крупно зернистыми.

Если часть элементов матрицы связности равны нулю, то сложная задача, состоящая из набора простых, называется слабо связанной. Такие задачи образуют обширный класс, в котором простые задачи объединяются либо функциональными связями, либо связями по данным. Принято считать, что общий объем взаимодействия по данным меньше объема вычисления, то появляющийся набор простых задач можно считать общей слабосвязанной задачей. Наличие связности между задачами не позволяет эффективно решать ее на распределенных устройствах. Объектом распараллеливания является набор вызываемых процедур. Организация управления вычисления определяется программистом. Такую организацию принято называть среднезернистым параллелизмом.

Сложная задача, состоящая из набора простых, называют сильно связанное, если все элементы имеют не нулевые значения. В сильносвязных задачах число обменных взаимодействий по управлению данных сопоставимо с объемом вычислением в простых задачах. В предельном случае

на каждом шаге вычислений выполняется обмен данными между простыми задачами. Распараллеливание выполняется на уровне внутренних блоков (циклов). Это мелкозернистая организация.

Методы и средства организации параллельных вычислений определяются уровнем организации параллельных вычислений. Традиционно принято выделять 4 уровня:

- Уровень заданий.
- Уровень программ.
- Уровень команд. Реализуется средствами аппаратного ядра ОС.
- Арифметический уровень. Обеспечивается параллельное или конвейерное исполнение команды. Реализуется аппаратными средствами.

С программной точки зрения параллельные программы для организации параллельных вычислений должны обладать 4 свойствами:

- Параллелизм. Способность выполнять несколько действий одновременно
- Масштабируемость. Возможность увеличения производительности, при увеличении числа процессоров.
- Локальность. Предпочтительность в использовании локальной памяти, к удаленной
- Модульность. Возможность декомпозиции сложных программ на простые компоненты.

Получаемое при этом время решение задачи должно быть оптимальным и приемлемым для заказчика.

2. Архитектура параллельных вычислительных систем

2.1. АРХИТЕКТУРЫ С РАЗДЕЛЯЕМОЙ ОБЩЕЙ ПАМЯТЬЮ

Один из наиболее важных классов параллельных машин – shared memory multiprocessors – многопроцессорные системы с разделяемой общей областью памяти. Ключевой характеристикой данных систем является то, что взаимодействие процессоров осуществляется как обычное выполнение инструкций доступа к памяти (т.е. loads and stores). Данный класс систем имеет большую историю развития, начало которой датируется 1960 годом (система BINAC).

Основа программной модели (Shared address) для таких архитектур, по существу, есть разделение времени доступа к общей области памяти (time – sharing). В процессах часть их адресного пространства является разделяемой с другими процессами. Каждый процесс имеет виртуальную область памяти, состоящую из адресного пространства разделяемой памяти и собственного адресного пространства. На рис. 1.1 представлена типовая модель взаимодействия процессоров через механизм разделяемой общей памяти. На рисунке показана связь виртуального адресного пространства процессов (P_0 – P_n) , состоящего из разделяемой и собственной областей, с физической областью памяти.

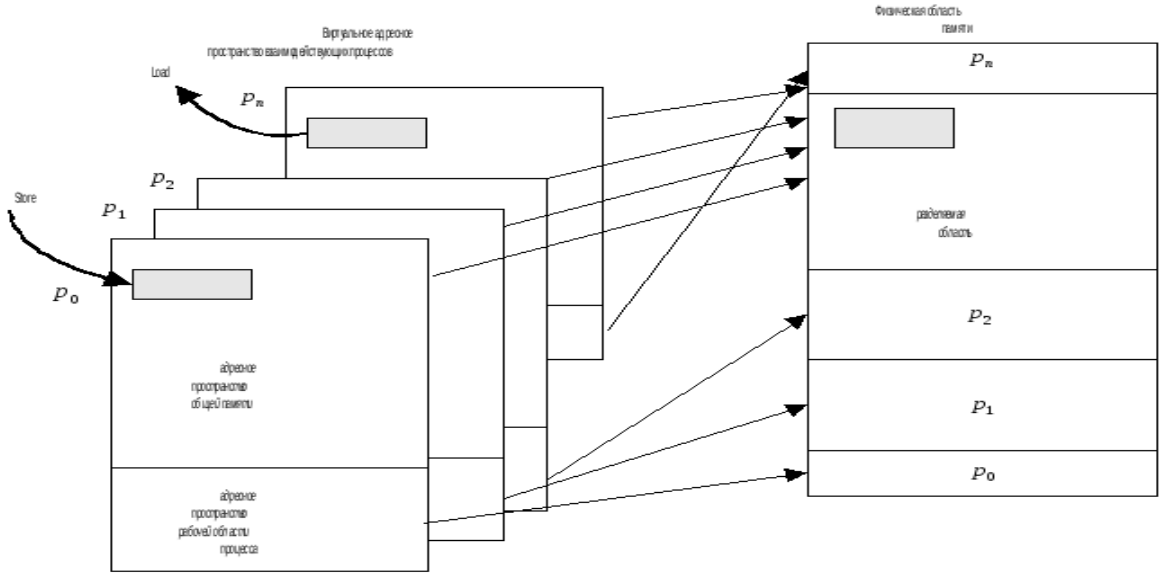


Рис. 1.1. Взаимодействие процессов в модели разделяемой общей памяти

Операции записи и чтения в разделяемую область памяти требует дополнительного контроля. Т.е. операционная среда выполняет специальные функции синхронизации процессов (операций записи и чтения в разделяемую область памяти). Например, должна быть блокирована операция чтения данных одного процесса до тех пор, пока в данную ячейку не будет записан результат процесса – предшественника.

Наращивание мощности системы достигается простым добавлением процессоров, модулей памяти и числа контроллеров ввода-вывода (которые также являются разделяемыми).

2.2. АРХИТЕКТУРЫ С РАСПРЕДЕЛЕННОЙ ОБЛАСТЬЮ ПАМЯТИ

Ко второму важному классу параллельных машин относятся многопроцессорные системы с распределенной областью памяти - *message-passing architectures* (MPA). MPA используют законченные компьютеры, включающие микропроцессор, память и подсистему ввода-вывода, как узлы для построения системы, объединенные коммуникационной средой, обеспечивающую взаимодействие процессоров посредством простых операций ввода-вывода. Структура высокого уровня для MPA практически такая же, как и для NUMA машин, т.е. машин с разделяемой памятью, показанных на рис. 1.6. Первое отличие состоит в том, что коммуникации интегрированы в уровень ввода-вывода, а не в систему доступа к памяти. Этот стиль дизайна имеет много общего с сетями из рабочих станций или кластерными системами, за исключением того, что в MPA пакетирование узлов обычно более плотное, нет монитора и клавиатуры на каждом узле, а производительность сети намного выше стандартной. Интеграция между процессором и сетью имеет склонность быть

более тесной чем традиционные структуры ввода-вывода, которые поддерживают соединения с оборудованием, которое более медленное, чем процессор. Начиная с посылки сообщения MPA есть фундаментальное взаимодействие ПРОЦЕССОР – ПРОЦЕССОР.

Системы с распределенной памятью имеют существенную дистанцию между программной моделью и действительными аппаратными примитивами. Коммуникации осуществляются через средства операционной системы или библиотеку вызовов, которые выполняют много акций более низкого уровня, включающих операции коммуникации.

Наиболее общие операции взаимодействия на пользовательском уровне (user-level) в MPA есть варианты посылки (SEND) и получения (RECEIVE) сообщения. Совместный механизм SEND и RECEIVE, вызванный передачей данных из одного процесса в другой, показан на рис 1.8.

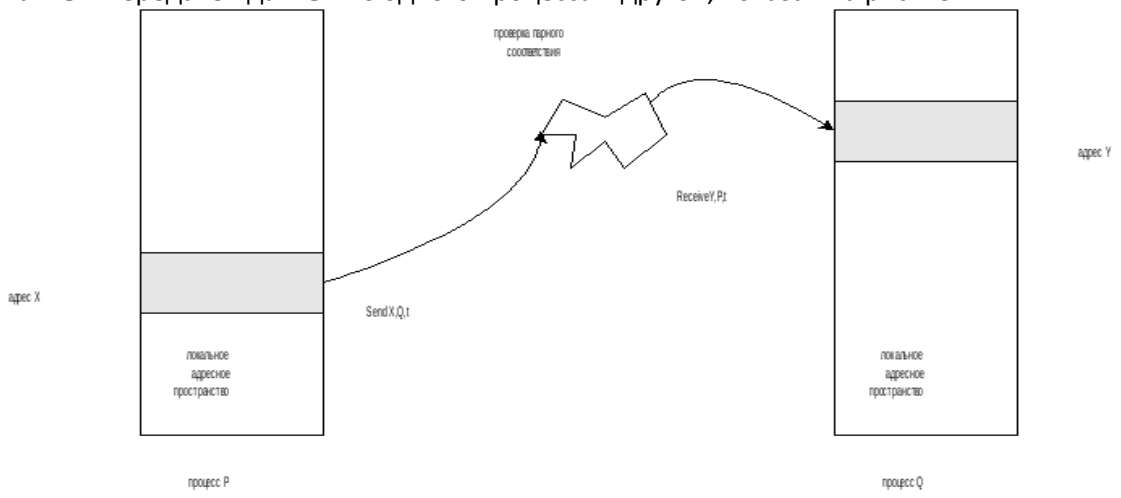


Рис. 1.8. Принцип взаимодействия процессов в системах с распределенной областью памяти

Передача данных из одного локального адресного пространства к другому произойдет, если посылка сообщения со стороны процесса - отправителя будет востребована процессом - получателем сообщения.

2.3. МАТРИЧНЫЕ СИСТЕМЫ

Третий важный класс параллельных машин – матричные системы - относятся к классу *single-instruction-multiple-data machines* (SIMD) или *data parallel architectures*. Ключевой характеристикой этой модели программирования *data parallel programming model* [2] является то, что операция может быть выполнена параллельно на каждом элементе большой регулярной структуры данных, таких как массивы и матрицы.

В этом случае обычные последовательные компьютеры определяются как системы с одиночным потоком команд и одиночным потоком данных *single-instruction-single-date* (SISD), а параллельные машины, построенные из множества обычных процессоров как системы с множественным потоком команд и множественным потоком данных *multiple-instruction-multiple-data* (MIMD). Альтернативой им стали системы класса одиночного потока команд и множественного потока данных *single-instruction-multiple-data* (SIMD).

В SIMD машинах модель программирования *(data parallel programming model)* была напрямую реализована в физическом аппаратном уровне. Обычно управляющий процессор (Пр) осуществлял рассылку каждой инструкции по массиву процессорных элементов *(data processing element)* (ПрЕ), которые были соединены между собой в форме регулярной решетки, как показано на рис 1.12

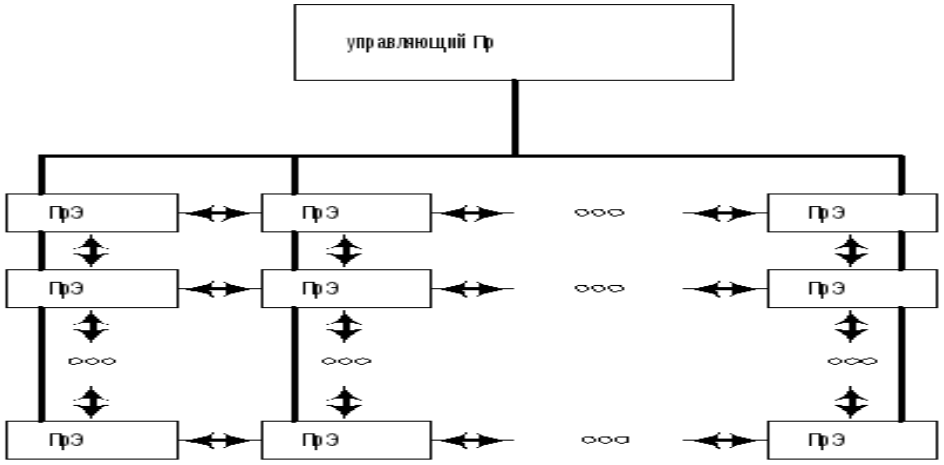


Рис. 1.12. Структура матричных систем

Матричные системы ориентированы на большой класс вычислительных задач, которые требуют выполнения матричных операций или обработки массивов, т.е. когда выполняются одинаковые операции на уровне каждого элемента массива или матрицы, вовлекающие в процесс вычисления соседние элементы. Т.о. управляющий процессор инструктирует процессоры данных по выполнению каждой операции над локальным элементом данных или все выполняют операцию коммуникации-обмена данных (все сразу).

2.4. МАШИНЫ, УПРАВЛЯЕМЫЕ ПОТОКОМ ДАННЫХ

Машины, управляемые потоком данных, относятся к классу *datafllow architecture*. Реализация *datafllow* - модели вычислений может привести к наивысшей степени параллелизма, т.к. в ней используется альтернативный принцип управления – управление потоком данных, который не накладывает дополнительных ограничений, присущих рассмотренному выше командному принципу управления.

В вычислительных системах, управляемых потоками данных, машинах потоков данных, отсутствует понятие программы как последовательности команд, а, следовательно, отсутствует понятие состояния процесса. Каждая инструкция передается на исполнение, как только создаются условия для ее реализации. При наличии достаточных аппаратных средств одновременно может обрабатываться произвольное число готовых к исполнению инструкций. В *datafllow* – модели параллелизм не задается явно и аппаратные средства обработки должны его выделять на этапе исполнения. Однако, следует отметить, что реализация принципа управления потоком данных вызывает ряд трудностей, часть из которых носит принципиальный характер. К их числу необходимо отнести громоздкость программы, трудность обработки итерационных циклов, трудность работы сос структурами данных.

Суть идеи *datafllow* - модели в том, что программа представляется графом потока данных, пример которого показан на рис. 1.13

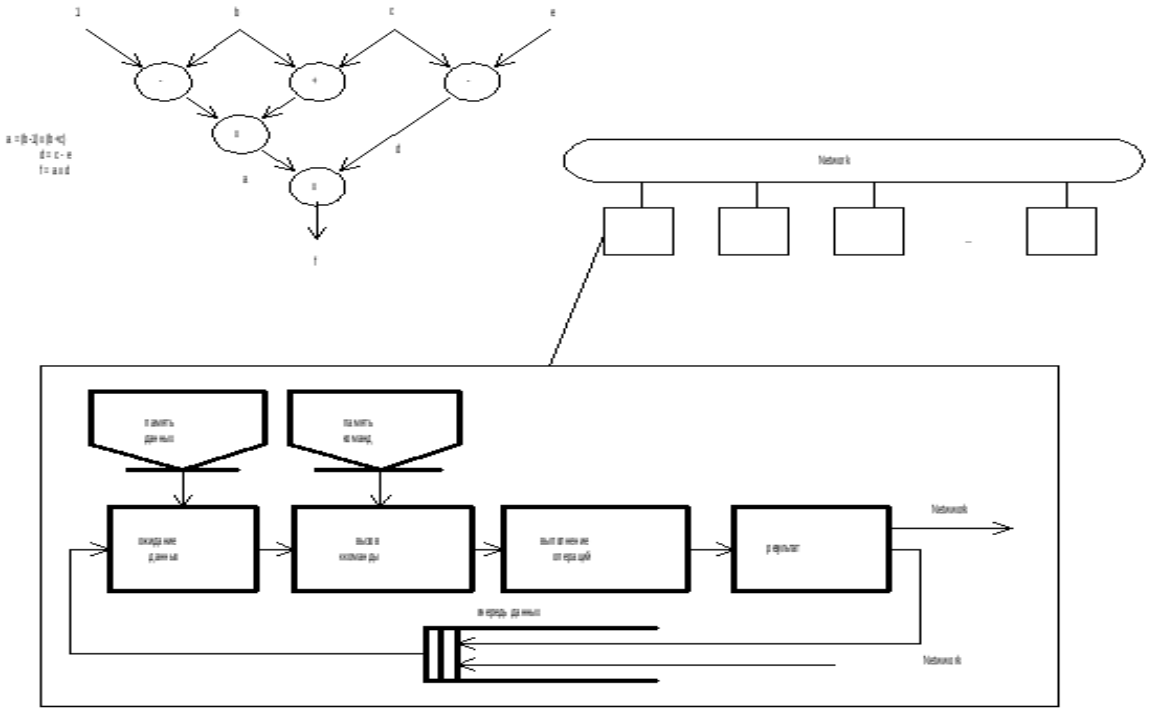


Рис. 1.13. Граф потока данных и структура потоковых машин

Инструкциям на графе соответствуют вершины, а дуги, обозначенные стрелками, указывают на отношения предшествования. Точка вершины, в которую входит дуга, называется входным портом (или входом), а точка, из которой она выходит, выходным. По дугам передаются метки, называемые токенами данных *(token)*. Срабатывание вершины означает выполнение инструкции. При этом срабатывание происходит в произвольный момент времени при выполнении условий, соответствующих правилу запуска. Обычно используется строгое правило запуска, согласно которому срабатывание вершины происходит при наличии хотя бы по одному токену во всех ее входных портах. Срабатывание вершины сопровождается удалением одного токена из каждого входного порта и размещением не более одного токена результата операции в выходные порты. В зависимости от конкретной архитектуры системы порты могут хранить один или несколько токенов, причем они могут использоваться по правилу FIFO или в произвольном порядке.

Граф может быть распространен на произвольную совокупность процессоров. В предельном случае процессор в машине, управляемой потоком данных, может выполнять операции как отдельный круговой поплайн, как показано на рис 1.13.

Токен или сообщение из сети содержит данные и адрес или тэг *(tag)* его места назначения (вершины графа). Тэг сравнивается с хранимыми тэгами на совпадение. Если совпадение не произошло, то токен помещается в память для ожидания партнера. Если партнер найден, то токен с совпавшим тэгом удаляется из памяти, и данные поступают на выполнение соответствующей инструкции. Когда результат вычислен, новое сообщение или токен, содержащий данные результата посылаются каждому по назначению, специфицированному в инструкции. Тот же самый механизм может быть использован и для удаленного процессора.

Все архитектуры машин, управляемых потоком данных, с точки зрения механизмов организации повторной входимости, принято делить на статические и динамические. Т.е. в них используется либо статический граф потока данных, где каждая вершина представлена примитивной операцией, либо динамический граф, в котором вершина может быть представлена вызовом произвольной функции, которая сама может быть представлена графом. В динамических или *(tagged-token)* архитектурах эффект динамически развивающегося графа на вызываемую функцию обычно достигается появлением дополнительного информационного контекста в тэге.

2.5 СИСТОЛИЧЕСКИЕ СИСТЕМЫ

Разработчики систолических архитектур ставили перед собой задачу получить систему, которая совмещала бы достоинства конвейерной и матричных обработок. Первоначально систолические архитектуры разрабатывались для узкоспециализированных вычислительных систем. Однако в дальнейшем были найдены соответствующие алгоритмы для достаточно широкого класса задач, позволяющие реализовать принципы систолической обработки.

Основной принцип систолической обработки заключается в том, чтобы выполнить все стадии обработки каждого элемента данных, извлеченного из памяти, прежде чем вновь поместить в память результат этой обработки. Этот принцип реализуется систолической матрицей процессорных элементов, в которой отдельные процессорные элементы объединены между собой прямыми и регулярными связями, образующими конвейер. Таким образом может формироваться несколько потоков операндов, каждый из которых образован исходными операндами (элементами структуры данных, хранящейся в памяти), промежуточными результатами, полученными при выполнении элементарных операций в каждом процессорном элементе, и элементами результирующей структуры. Потоки данных синхронизированы единой для всех процессорных элементов системой тактовых сигналов. Во время тактового интервала все элементы выполняют короткую неизменную последовательность команд (или одну команду).

$$y(i) = w1 \times x(i) + w2 \times x(i+1) + w3 \times (i+2) + w4 \times x(i+3)$$

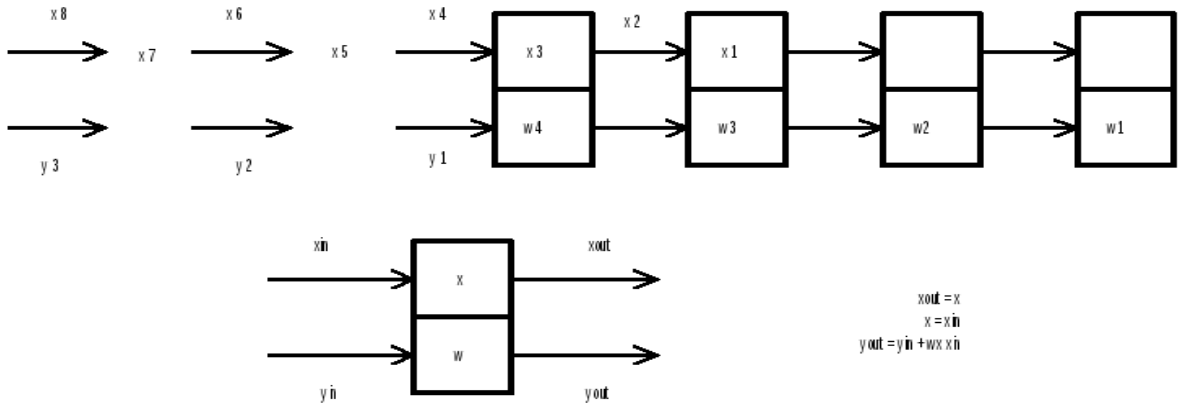


Рис. 1.14. Пример построения систолической структуры

Рис 1.14 иллюстрирует систолическую систему для вычисления «свертки функции», использующую простую линейную область. Во время тактового интервала входные данные продвигаются вправо, умножаются на локальный вес и аккумулируются на выходе в порядке следования.

Систолические архитектуры обладают следующими достоинствами:

минимизируются обращения к памяти, сто позволяет согласовать скорость работы памяти со скоростью обработки, упрощается решение проблем ввода – вывода вследствие уменьшения конфликтов при обращении к памяти, эффективно используются технологические возможности СБИС за счет регулярности структуры систолической матрицы.

3. Алгоритмические структуры: параллельная, групповая…

3.1. Последовательная организация

В обычной последовательной алгоритмической структуре (рис.3.1.*а*) в каждый момент времени исполняется только одна команда. Соответствующая вычислительная машина проста: в ней имеется единственная память для хранения данных и программ, одно арифметическое устройство, исполняющее текущую команду, одно устройство управления, осуществляющее контроль за исполнением, счетчик команд, хранящий адрес текущей команды, и простой механизм его модификации. Взаимные связи между перечисленными компонентами также просты. Быстродействие системы ограничено ее последовательной сущностью и определяется временем исполнения каждой команды.

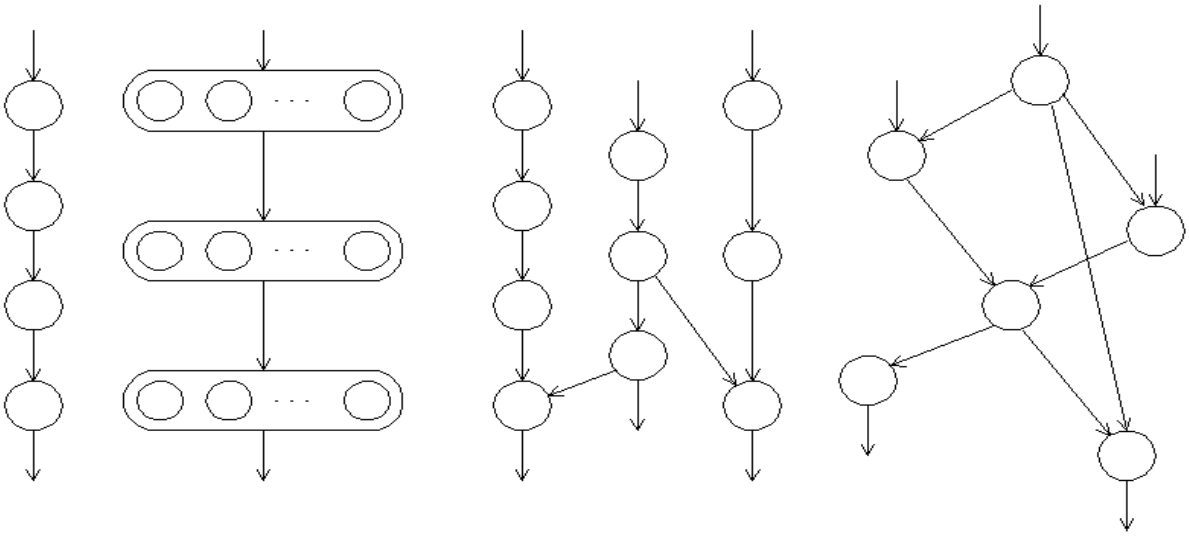


Рис.3.1. Алгоритмические структуры: *а* - последовательная; *б* - последовательно групповая; *в* - совокупность слабосвязанных потоков; *г* - параллельная структура общего вида.

Важное преимущество последовательного подхода состоит в том, что для машин данного типа разработано огромное количество алгоритмов и программ, накоплен богатый опыт программирования, разработаны фундаментальные языки и технологии программирования. По этой причине дальнейшее совершенствование организации шло так, что последовательная структура сохранялась, однако к ней добавлялись механизмы, позволяющие исполнять несколько команд одновременно. Все эти механизмы так или иначе сводятся к общему принципу *опережающего просмотра команд*, поскольку для того, чтобы иметь возможность одновременно исполнять несколько команд, процессор должен просматривать поток команд на несколько шагов вперед и находить те команды, которые допускают одновременное выполнение. Выявление параллелизма в этом случае сводится к тому, чтобы исключить те зависимости, которые привнесены в алгоритм в процессе его последовательной записи, и сохранить только те зависимости, которые обусловлены самой природой алгоритма. Такие зависимости бывают двух типов: *зависимости по данным* (когда команда не может исполниться до тех пор, пока не подготовлены все операнды, которые являются результатами исполнения других команд) и *зависимости по управлению* (когда условный переход не может быть осуществлен до тех пор, пока не вычислено условное выражение).

Для одновременного выполнения команд, не являющихся зависимыми, существуют две дополнительные возможности: использование нескольких процессоров (по числу команд) и применение одного процессора конвейерного типа. Конвейеризация обработки - это общий метод повышения пропускной способности систем, выполняющих повторяющиеся операции. Основа этого подхода состоит в том, что система может быть разделена на ступени обрабатывающих устройств, позволяющих начинать исполнение новой команды прежде, чем будет завершена предыдущая. В такой системе пропускная способность определяется временем прохождения самого медленного звена. Данный подход может быть применен, например, к обработке команд,

поскольку процесс исполнения команды состоит из нескольких стадий, таких как вызов команды, расшифровка, вычисление адреса, вызов операнда и др.
Факторы, от которых зависит ускорение вычислений за счет опережающего просмотра, следующие:
- степень параллелизма, заключенного в программе. В типовых программах число зависимостей по данным невелико. Это создает широкие предпосылки для использования параллелизма. Но с другой стороны, число зависимостей по управлению, соответствующих условным переходам, весьма значительно, и оно накладывает ограничения, но возможность использования параллелизма. В некоторых вычислительных машинах имеется механизм, который не останавливается на условных переходах, а продолжает опережающий просмотр по обеим ветвям или по предпочтительной ветви, определяемой с помощью механизма прогнозирования перехода. Этот метод требует сложного дополнительного оборудования в устройстве управления, и в то же время имеет ограниченный эффект из-за высокой вероятности того, что еще до завершения предыдущего ветвления появится новая ветвь.
  - способность устройства управления обнаруживать зависимости. Если поставлена задача обнаружить все или почти все операции, допускающие одновременное исполнение, то устройство управления получается довольно сложным.
  - наличие конвейерного процессора и/или набора функциональных устройств, предназначенных для одновременного исполнения нескольких команд.
  - наличие памяти с широким трактом доступа, способным обеспечить поставку команд и данных процессору в таком темпе, чтобы он оказывался постоянно активным. Поскольку быстродействие памяти обычно ниже, чем процессора, то для организации широкого тракта доступа необходимо разбить память на блоки, обращение к которым может происходить одновременно. При этом максимальная пропускная способность тракта, связывающего процессор с памятью, зависит от быстродействия блоков и их общего количества. Однако, поскольку возможны конфликты по доступу, пропускная способность оказывается зависящей и от других факторов, таких как механизм адресации, адресный поток, а также возможность приоритетного обслуживания запросов.

Еще один подход к уменьшению отрицательного влияния на производительность зависимостей по данным и по управлению состоит в том, что процессор работает как бы в мультиплексном (по времени) режиме, обслуживая несколько процессов. Этот принцип похож на используемый в мультипрограммировании, с той лишь разницей, что в данном случае мультиплексирование осуществляется с квантом, равным времени выполнения одной команды.
3.2. Последовательно групповая организация
В последовательно групповой структуре вычислительные конструкции, образующие алгоритм, объединены в группы, а отношение следования состоит в том, что конструкции внутри группы могут выполняться одновременно, а сами группы последовательно (рис. 2.1.*б*)
3.3. Параллельная структура общего вида
При значительном увеличении частоты взаимодействия между потоками характер алгоритмической структуры совершенно меняется - она превращается в параллельную структуру общего вида (см. рис.3.1г). Такая структура содержит в себе наибольшие возможности для организации параллельных вычислений и увеличения скорости обработки, но в тоже время, она таит серьезные трудности, связанные с представлением программ в системе, определением последовательности выполнения операций и накладными расходами на синхронизацию. Для параллельных структур общего вида практически невозможно явно задать последовательность выполнения команд в объектном коде, поэтому возникает необходимость с помощью указателей задавать отношение предшествования на множестве операторов, что значительно усложняет представление программ.
Далее из-за того, что в условие инициализации операции входит завершение предшествующих операций, управление порядком исполнения команд становится более сложным. Для осуществления такого управления предложено два механизма:
1. С помощью потоков признаков.

2. С помощью потоков данных.

Ряд проблем при реализации механизма управления с помощью потоков данных возникает в тех случаях, когда алгоритм содержит либо циклы (т.е. когда какую-либо группу команд необходимо исполнить несколько раз подряд), либо подпрограммы. Существуют две группы способов реализации потокового управления:

- способы, основанные на *статическом* подходе, когда циклы и подпрограммы раскрываются на этапе трансляции, так что в результате каждая команда должна исполняться только один раз,
- способы, основанные на *динамическом* подходе, когда операнды снабжаются метками и группируются в пары таким образом, чтобы для использования разных реализаций тела цикла (или подпрограммы) оказывалось возможным генерировать копию одной и той же команды.

Преимущество статического подхода состоит в простоте управления последовательностью операций, поскольку в этом случае принцип потокового управления реализуется в чистом виде - команды исполняются сразу, как только поступают операнды. В динамическом случае, прежде чем воспроизвести единственную копию команды, необходимо собрать вместе операнды, помеченные одной и той же меткой. Это значительно усложняет реализацию данного подхода. С другой стороны, динамический вариант позволяет получать менее объемные программы и к тому же реализовывать параллелизм, появляющийся в процессе счета (например, при исполнении рекурсивных процедур или циклов, зависящих от данных).

4. Факторы ускорения вычислений. Синтаксическая и семантическая векторизация

Понятие “фактор ускорения” (“speedup factor”) - это отношение времени выполнения лучшего последовательного алгоритмам к времени выполнения параллельного алгоритма.

S(p) = T1 / Tp

Теоретическую оценку максимального ускорения, достижимого для параллельного алгоритма с долей последовательных операций равной α определяется законом Амдала:

$$S = \frac{T_1}{T_p} = \frac{T_1}{\alpha T_1 + \frac{(1-\alpha)T_1}{p}} \leq \frac{1}{\alpha}.$$

Таким образом, если всего 10% операций алгоритма не может быть выполнена параллельно, то никакая параллельная реализация данного алгоритма не может дать больше ускорение более чем в 10 раз.

Векторизация (в параллельных вычислениях) — вид распараллеливания программы, при котором однопоточные приложения, выполняющие одну операцию в каждый момент времени, модифицируются для выполнения нескольких однотипных операций одновременно.

Скалярные операции, обрабатывающие по паре операндов, заменяются на операции над массивами (векторами), обрабатывающие несколько элементов вектора в каждый момент времени.

Наиболее общим типом векторизации является синтаксическая векторизация, т.е. такой способ преобразования последовательности команд, при котором не учитывается смысловое значение команд и данных, а учитывается только их форма. Самая сильная сторона этого типа одновременно является и его самой слабой. С одной стороны, поскольку учитывается только форма представления программ и данных, имеется теоретическая возможность реализовать процесс векторизации с помощью специальных программных средств - так называемых автоматических векторизаторов. Но с другой стороны, что же делать, если эти программные векторизаторы не найдут формального преобразования, позволяющего превратить некоторую программно-информационную структуру в векторную?

Для того, чтобы избежать данной ситуации, необходимо обратиться к другому подходу - семантической векторизации. Семантическая векторизация подразумевает интерпретацию содержащихся в программах структур данных и подстановку векторных конструкций, выполняемых с учетом смыслового содержания задачи, а не ее формы.

5. Основные модели межпроцессорного общения: передача сообщений и общая память

Способы межпроцессного обмена.
Традиционно считается, что основными способами межпроцессного обмена являются каналы и разделяемая память.

Механизмы:

- механизмы обмена сообщениями;
- механизмы синхронизации;
- механизмы разделения памяти;
- механизмы удалённых вызовов (RPC).

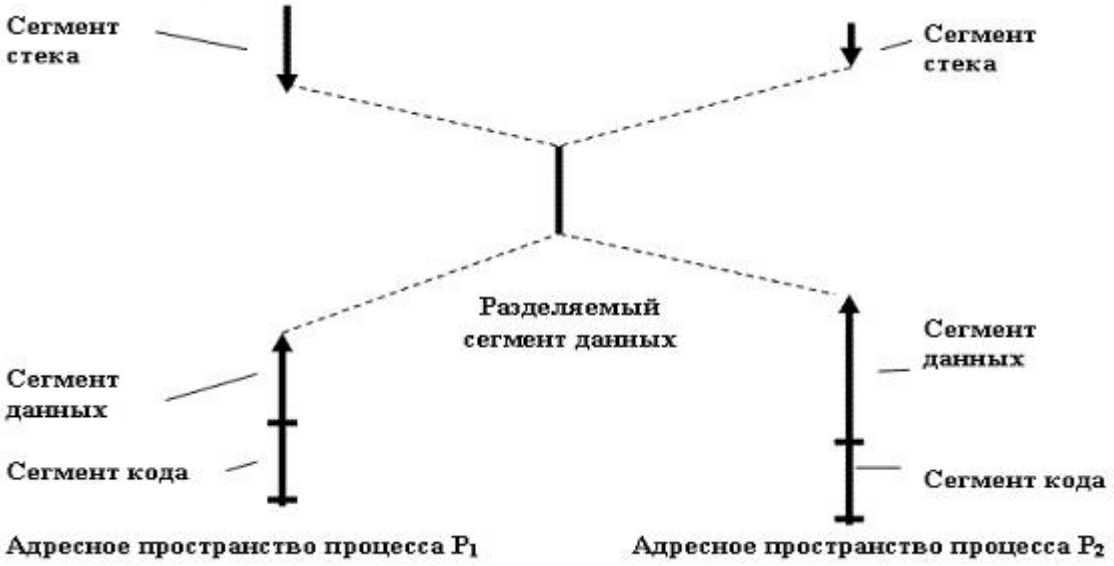
При передаче в рамках потоковой модели данные представляют собой неструктурированную последовательность байтов и никак не интерпретируются системой. В модели сообщений на передаваемые данные накладывается некоторая структура, обычно их разделяют на сообщения заранее оговоренного формата.

Понятие о разделяемом ресурсе
Разделяемую память применяют для того, чтобы увеличить скорость прохождения данных между процессами. В обычной ситуации обмен информацией между процессами проходит через ядро. Техника разделяемой памяти позволяет осуществить обмен информацией не через ядро, а используя некоторую часть виртуального адресного пространства, куда помещаются и откуда считываются данные.
После создания разделяемого сегмента памяти любой из пользовательских процессов может подсоединить его к своему собственному виртуальному пространству и работать с ним, как с обычным сегментом памяти. Недостатком такого обмена информацией является отсутствие каких бы то ни было средств синхронизации, однако для преодоления этого недостатка можно использовать технику семафоров.
Примерный сценарий использования разделяемой памяти при реализации технологий «клиент—сервер» имеет вид:

1) сервер получает доступ к разделяемой памяти, используя семафор;
2) сервер производит запись данных в разделяемую память;
3) после завершения записи данных сервер освобождает доступ к разделяемой памяти с помощью семафора;
4) клиент получает доступ к разделяемой памяти, запирая доступ к этой памяти для других процессов с помощью семафора;
5) клиент производит чтение данных из разделяемой памяти, а затем освобождает доступ к памяти с помощью семафора.

Межпроцессный обмен базируется на разделяемых ресурсах, к которым имеет доступ некоторое множество процессов. При этом возникают задачи создания, именования и защиты таких ресурсов. Обычно один из процессов создает ресурс, наделяет его атрибутами защиты и именем, по которому

данный ресурс может быть доступен остальным процессам (даже в случае завершения работы процесса-создателя).



Каналы связи

Основной принцип работы канала состоит в буферизации вывода одного процесса и обеспечении возможности чтения содержимого программного канала другим процессом. При этом часто интерфейс программного канала совпадает с интерфейсом обычного файла и реализуется обычными файловыми операциями read и write. Для обмена могут использоваться потоковая модель и модель обмена сообщениями.

Механизм генерации канала предполагает получение процессом-создателем (процессом-сервером) двух описателей для пользования этим каналом. Один из описателей применяется для чтения из канала, другой - для записи в канал.

Процесс, создающий канал, принято называть сервером, а другой процесс - клиентом. Для общения с каналом клиент и сервер должны иметь описатели (дескрипторы, handles) для чтения и записи. Процесс-сервер получает описатель при создании канала. Процесс-клиент может получить описатели в результате наследования, в том случае, когда клиент является потомком сервера. Это типично для общения через так называемые анонимные каналы. Другой способ получения - открытие по имени уже существующего именованного канала неродственным процессом, который в результате также становится обладателем необходимых описателей.

6. Модели параллельного программирования

В настоящее время в области научно-технических вычислений превалируют три модели параллельного программирования: модель передачи сообщений, модель с общей памятью и модель параллелизма по данным.

SPMD (single program, multiple data) — описывает систему, где на всех процессорах MIMD-машины выполняется только одна единственная программа, и на каждом процессоре она обрабатывает разные блоки данных.

MPMD (multiple programs, multiple data) — описывает систему, а) где на одном процессоре MIMD-машины работает мастер-программа, а на других подчиненная программа, работой которой руководит мастер-программа (принцип *master/slave* или *master/worker*); б) где на разных узлах MIMD-машины работают разные программы, которые по-разному обрабатывают один и тот же массив данных, большей частью они работают независимо друг от друга, но время от времени обмениваются данными для перехода к следующему шагу.

Процесс/канал (*Process/Channel*)
В этой модели программы состоят из одного или более процессов, распределенных по процессорам. Процессы выполняются одновременно, их число может измениться в течение времени выполнения программы. Процессы обмениваются данными через каналы, которые представляют собой однонаправленные коммуникационные линии, соединяющие только два процесса. Каналы можно создавать и удалять.

Модель передачи сообщений

- Программа порождает несколько задач.
- Каждой задаче присваивается свой уникальный идентификатор.
- Взаимодействие осуществляется посредством отправки и приема сообщений.
- Новые задачи могут создаваться во время выполнения параллельной программы, несколько задач могут выполняться на одном процессоре.

Модель параллелизма данных

- Одна операция применяется к множеству элементов структуры данных. Программа содержит последовательность таких операций.
- "Зернистость" вычислений мала.
- Программист должен указать транслятору, как данные следует распределить между задачами.

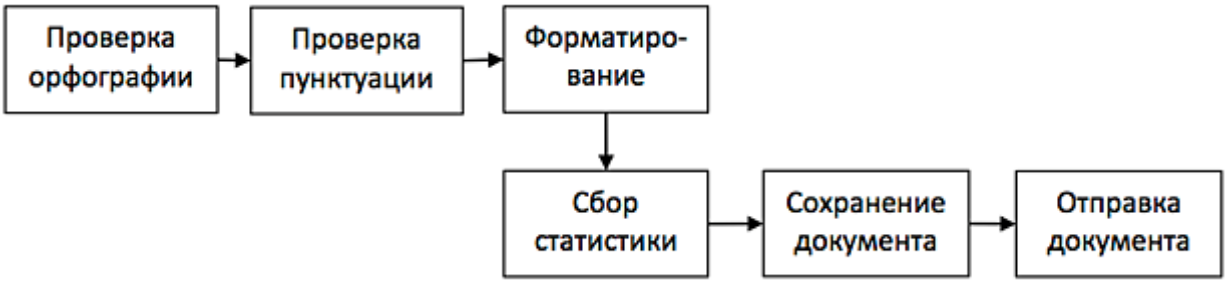
Модель общей памяти
В модели общей (разделяемой) памяти задачи обращаются к общей памяти, имея *общее адресное пространство* и выполняя операции считывания/записи. Управление доступом к памяти осуществляется с помощью разных механизмов, таких, например, как семафоры. В рамках этой модели не требуется описывать обмен данными между задачами в явном виде. Это упрощает программирование. Вместе с тем особое внимание приходится уделять соблюдению детерминизма, таким явлениям, как "гонки за данными" и т. д.

Параллелизм задач
В данной модели идея основана на разбиении вычислительной задачи на несколько относительно самостоятельных подзадач. Каждая подзадача выполняется на своем процессоре. Данный подход ориентирован на архитектуру *MIMD*.

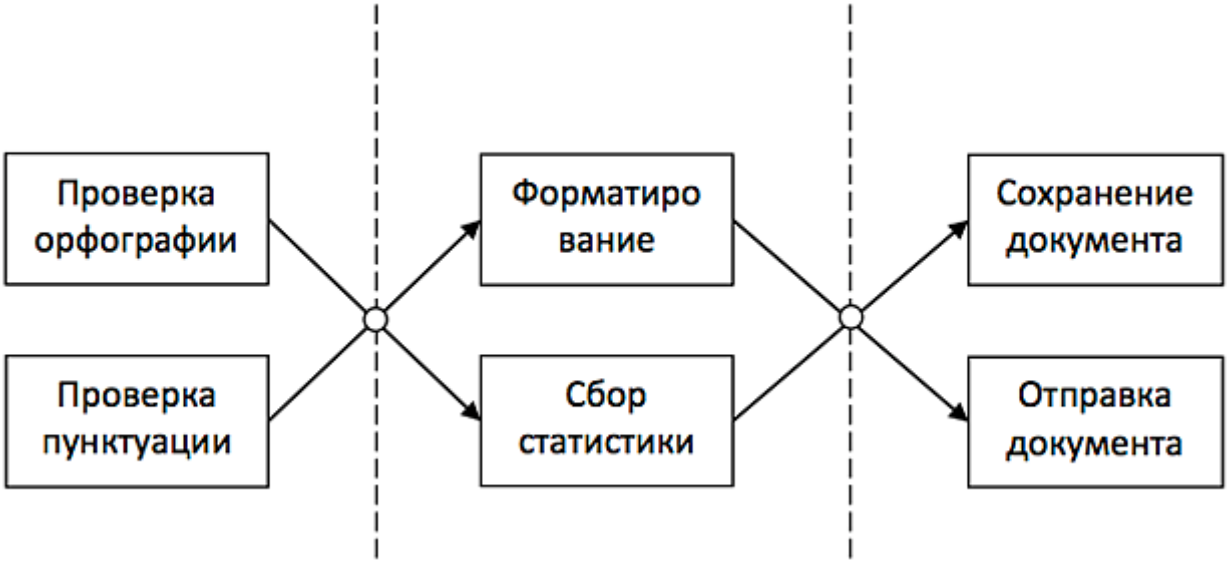
7. Декомпозиция данных и функциональная декомпозиция

Декомпозиция

Под декомпозицией понимается разбиение задачи на относительно независимые части (подзадачи). Декомпозиция задачи может быть проведена несколькими способами: по заданиям, по данным, по информационным потокам.
Декомпозиция по заданиям (функциональная декомпозиция) предполагает присвоение разным потокам разных функций. Например, приложение выполняющее правку документа включает следующие функции: проверка орфографии CheckSpelling, проверка пунктуации CheckPuncto, форматирование текста в соответствие с выбранными стилями Format, подсчет статистики по документу CalcStat, сохранение изменений в файле SaveChanges и отправка документа по электронной почте SendDoc.



Функциональная декомпозиция разбивает работу приложения на подзадачи таким образом, чтобы каждая подзадача была связана с отдельной функцией. Но не все операции могут выполняться параллельно. Например, сохранение документа и отправка документа выполняются только после завершения всех предыдущих этапов. Форматирование и сбор статистики могут выполняться параллельно, но только после завершения проверки орфографии и пунктуации.



Декомпозиция по информационным потокам выделяет подзадачи, работающие с одним типом данных. В рассматриваемом примере могут быть выделены следующие подзадачи:

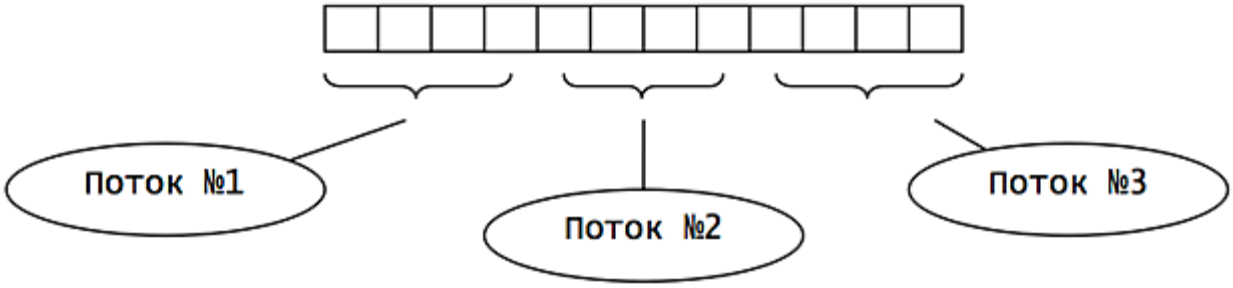
1. Работа с черновым документом (орфография и пунктуация);
2. Работа с исправленным документом (форматирование и сбор статистики);
3. Работа с готовым документом (сохранение и отправка).

При декомпозиции по данным каждая подзадача работает со своим фрагментом данных, выполняя весь перечень действий. В рассматриваемом примере декомпозиция по данным может применяться к задачам, допускающим работу с фрагментом документа. Таким образом,

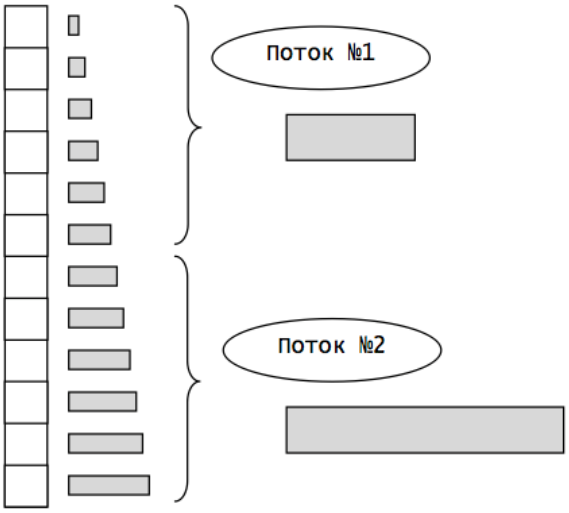
функции CheckSpelling, CheckPuncto, CalcStat, Format объединяются в одну подзадачу, но создается несколько экземпляров этой подзадачи, которые параллельно работают с разными фрагментами документа. Функции SaveChanges и SendDoc составляют отдельные подзадачи, так как не могут работать с частью документа.

Декомпозиция по данным

При декомпозиции по данным каждая подзадача выполняет одни и те же действия, но с разными данными. Во многих задачах, параллельных по данным, существует несколько способов разделения данных. Например, задача матричного умножения допускает разделение по строчкам – каждый поток обрабатывает одну или несколько строчек матрицы, по столбцам – каждый потокобрабатывает один или несколько столбцов, а также по блокам заданного размера.
Два основных принципа разделения данных между подзадачами – статический и динамический. При статической декомпозиции фрагменты данных назначаются потокам до начала обработки и, как правило, содержат одинаковое число элементов для каждого потока. Например, разделение массива элементов может осуществляться по равным диапазонам индекса между потоками (rangepartition).

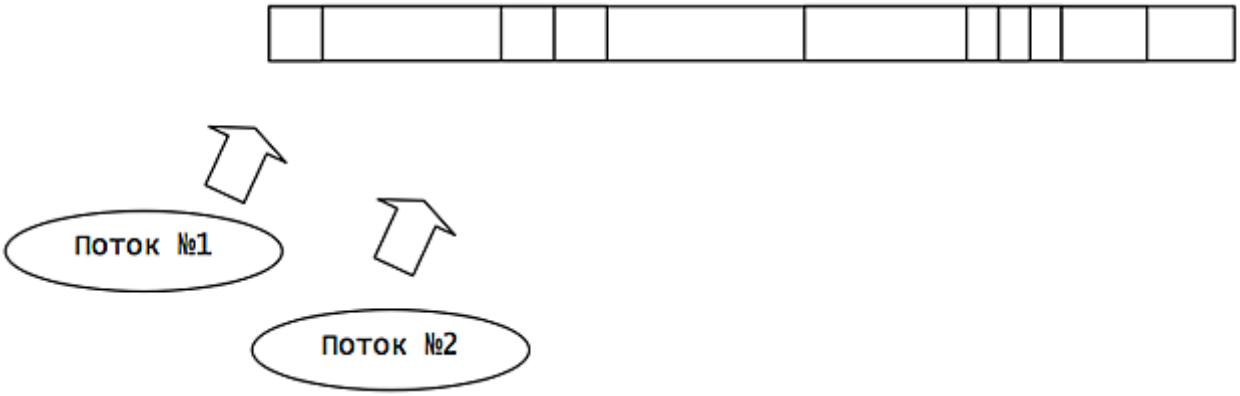


Основным достоинством статического разделения является независимость работы потоков (нет проблемы гонки данных) и, как следствие, нет необходимости в средствах синхронизации для доступа к элементам. Эффективность статической декомпозиции снижается при разной вычислительной сложности обработки элементов данных. Например, вычислительная нагрузка обработки i-элемента может зависеть от индекса элемента.



Разделение по диапазону приводит к несбалансированности нагрузки разных потоков. Несбалансированность нагрузки снижает эффективность распараллеливания. В некоторых случаях декомпозиция может быть улучшена и при статическом разбиении, когда заранее известно какие элементы обладают большей вычислительной трудоемкостью, а какие меньшей. Например, в случае зависимости вычислительной трудоемкости от индекса элемента, применение круговой декомпозиции выравнивает загруженность потоков. Первый поток обрабатывает все четные элементы, второй поток обрабатывает все нечетные.

В общем случае, когда вычислительная сложность обработки элементов заранее не известна, сбалансированность загрузки потоков обеспечивает динамическая декомпозиция.



/ При динамической декомпозиции каждый поток, участвующий в обработке, обращается за блоком данных (порцией). После обработки блока данных поток обращается за следующей порцией. Динамическая декомпозиция требует синхронизации доступа потоков к структуре данных. Размер блока определяет частоту обращений потоков к структуре. Некоторые алгоритмы динамической декомпозиции увеличивают размер блока в процессе обработки. Если поток быстро обрабатывает элементы, то размер блока для него увеличивается.

8. Мелкоблочный, среднеблочный и крупноблочный параллелизм

Функциональная декомпозиция - вначале сегментируется вычислительный алгоритм, а затем уже под эту схему подгоняется схема декомпозиции данных. Метод функциональной декомпозиции может оказаться полезным в ситуации, где нет структур данных, которые очевидно могли бы быть распараллелены.
Эффективность декомпозиции обеспечивается выполнением следующих рекомендаций:

- количество подзадач после декомпозиции должно примерно на порядок превосходить количество процессоров;
- следует избегать лишних вычислений и пересылок данных;
- подзадачи должны быть примерно одинакового размера;
- в идеале сегментация должна быть такой, чтобы с увеличением объема задачи количество подзадач также возрастало (при сохранении постоянным размера одной подзадачи).

Размер подзадачи определяется зернистостью алгоритма. Мерой зернистости является количество операций в блоке. Выделяют три степени зернистости:

1. Мелкозернистый параллелизм - на уровне команд (менее 20 команд на блок, количество параллельно выполняемых подзадач - от единиц до нескольких тысяч, средний масштаб параллелизма около 5 команд на блок).
2. Среднеблочный параллелизм - на уровне процедур. Размер блока до 2000 команд. Выявление такого параллелизма сложнее реализовать, поскольку следует учитывать межпроцедурные зависимости. Требования к коммуникациям меньше, чем в случае параллелизма на уровне команд.
3. Крупноблочный параллелизм - на уровне программ (задач). Соответствует выполнению независимых программ на параллельном компьютере. Крупноблочный параллелизм требует поддержки операционной системой.

Важнейшим условием декомпозиции является независимость подзадач.
Существуют следующие виды независимости:

- Независимость по данным - данные, обрабатываемые одной частью программы, не модифицируются другой ее частью.
- Независимость по управлению - порядок выполнения частей программы может быть определен только во время выполнения программы (наличие зависимости по управлению предопределяет последовательность выполнения).
- Независимость по ресурсам - обеспечивается достаточным количеством компьютерных ресурсов.
- Независимость по выводу - возникает, если две подзадачи не производят запись в одну и ту же переменную, а независимость по вводу-выводу, если операторы ввода/вывода двух или нескольких подзадач не обращаются к одному файлу (или переменной).

Полной независимости добиться обычно не удается.

9. Проектирование коммутационных обменов

Методы коммутации (переключения)

Важнейшим в организации работы коммуникационной сети является метод переключения, определяющий, как сообщения передаются от узла-источника к узлу-приемнику.

Сообщением называют логически завершенную порцию данных — пересылаемый файл, запрос на пересылку файла и т. д. Сообщения могут иметь любую длину. *Пакет* представляет собой часть сообщения, его длина ограничена, хотя и может варьироваться. Пакеты перемещаются по сети независимо друг от друга. Каждый пакет содержит заголовок с информацией об адресе получателя и его номер, что необходимо для правильной "сборки" пакета и восстановления целого сообщения.

Выделяют три основных метода коммутации в коммуникационных сетях параллельных вычислительных систем:

- коммутация с промежуточным хранением ("хранение-и-передача" — Store-and- Forward);
- коммутация цепей;
- метод виртуальных каналов.

Исторически, одним из первых методов коммутации был метод с промежуточным хранением (метод коммутации пакетов). В этом случае сообщение полностью принимается на каждом промежуточном узле и только после этого отправляется дальше. Пересылка выполняется, как только освобождается очередной канал передачи, тогда сообщение отправляется на следующий узел. Целое сообщение разбивается на пакеты, которые при достижении очередного узла сохраняются в специальном буфере. Буферизация требует дополнительной памяти и затрат времени. Задержка передачи пропорциональна расстоянию между источником и адресатом. Данный метод применяется в ситуациях, когда время отклика не имеет значения.

При коммутации цепей (коммутации каналов) между источником и получателем сообщения устанавливается непрерывная цепь, состоящая из отдельных каналов связи, проходящих через промежуточные узлы. После этого выполняется передача данных. Коммутируемый канал устанавливается только на время соединения отправителя и адресата.

Уменьшить задержки при передаче данных позволяет метод виртуальных каналов. В этом случае пакеты накапливаются в промежуточных узлах только тогда, когда недоступен очередной канал связи. В противном случае, пересылка выполняется немедленно и без буферизации.

10. Укрупнение блоков. Планирование вычислений. Балансировка нагрузки

1. Масштабирование разработанной параллельной вычислительной схемы необходимо проводить в случае, если количество имеющихся подзадач отличается от числа используемых процессоров. В случае нехватки процессоров следует выполнить укрупнение или агрегацию вычислений. Необходимо следить, чтобы получившиеся в результате укрупнения задачи имели одинаковую вычислительную сложность, а объем и интенсивность информационных взаимодействий между задачами был небольшим и не возрастал в результате проведенной агрегации. Очевидно, что первыми претендентами на объединение становятся задачи, связанные информационными зависимостями больше других.
При небольшом количестве имеющихся подзадач и существенно большем количестве доступных процессоров необходимо произвести детализацию или декомпозицию вычислений. При этом также необходимо учитывать вычислительную сложность получившихся в результате декомпозиции подзадач и информационные зависимости между ними. Как правило, проведение детализации вычислений не вызывает каких-либо существенных затруднений. Для примера поиска минимального значения в матрице агрегация вычислений может состоять в объединении нескольких столбцов для обработки на одном процессоре, а декомпозиция – в разбиении столбцов на несколько частей.

2. Распределение задач между процессорами является завершающим этапом разработки параллельного метода решения задачи. Управление распределением нагрузки для процессоров возможно только для вычислительных систем с распределенной памятью, в системах с общей памятью распределение нагрузки осуществляет операционная система.

3. Вопрос балансировки вычислительной нагрузки существенно усложняется в случаях, когда схема вычислений может существенно изменяться в ходе решения задачи. Причинами такого изменения могут быть: появившаяся в процессе расчета неоднородность сетки при решении уравнений в частных производных, разреженность матрицы, сингулярность интегралов, различные коэффициенты ветвления в задачах поиска и т. п. Кроме того, возможно изменение количества подзадач в процессе вычислений, в подобных случаях может потребоваться перераспределение подзадач между процессорами в ходе выполнения программы. В таких случаях говорят, что используется динамическая балансировка вычислительной нагрузки

Один из распространенных способов вычисления нагрузки основан на схеме “менеджер-исполнитель”. Для осуществления данного подхода необходимо, чтобы подзадачи, которые могут возникать и завершаться в процессе вычислений, не имели информационных взаимодействий или эти взаимодействия были бы минимальны. Согласно этой схеме, в системе выделяется один процессор-менеджер, который владеет информацией обо всех выполняющихся на процессорах-исполнителях подзадачах исходной задачи.

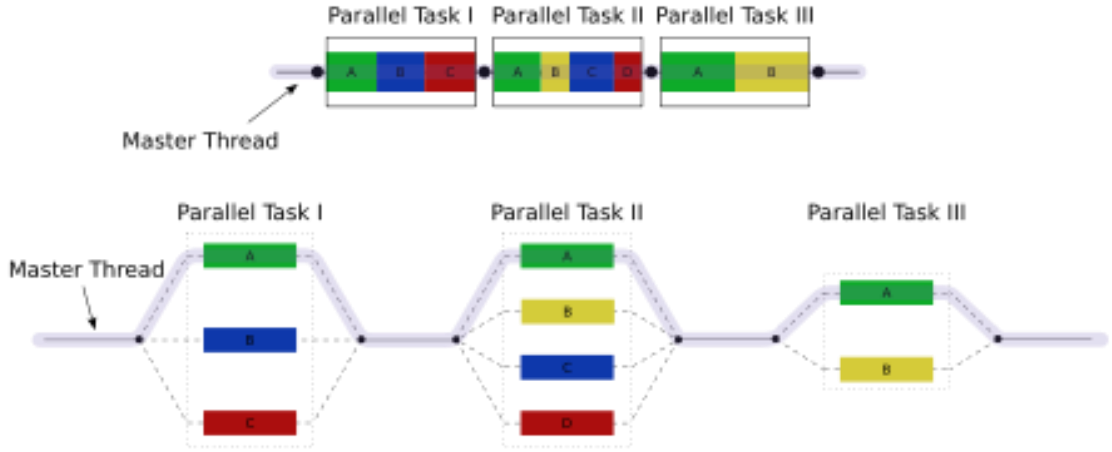
Процессоры-исполнители обращаются к процессору-менеджеру для получения очередной подзадачи, возникающие новые подзадачи передаются также процессору-менеджеру и распределяются на счет процессорам-исполнителям по мере их освобождения. Завершение вычислений происходит, когда процессоры-исполнители завершили решение выданных им подзадач, а у процессора-менеджера нет новых подзадач для вычисления.

1. Оценить правильность выполнения этапа планирования вычислений можно с помощью следующих вопросов:
2. Возможно ли избежать роста дополнительных вычислительных затрат при распределении нескольких задач на один процессор?
3. Существует ли необходимость динамической балансировки вычислений?
4. Не является ли процессор-менеджер "узким" местом при реализации схемы "менеджер-исполнитель"?

11. Основные приемы и способы разработки параллельных программ с применением технологии OpenMP.

OpenMP - API, предназначенное для программирования многопоточных приложений на многопроцессорных системах с **общей памятью**. OpenMP **поддерживается основными компиляторами**.

Программист сообщает компилятору с помощью директив `#pragma`, что блок кода может быть распараллелен. Зная данную информацию, компилятор в состоянии сгенерировать приложение, состоящее из одного главного потока, который создаёт множество других потоков для параллельного блока кода.



Использование OpenMP

```
gcc -fopenmp
Intel -openmp (Linux, MacOSX), -Qopenmp (Windows)
Microsoft -openmp (Настройки проекта в Visual Studio)
```

Синтаксис

Директивы OpenMP начинаются с #pragma omp.

`parallel`

Данная директива **создаёт группу из N потоков**. N определяется во время выполнения, обычно это число ядер процессора, но также можно задать N вручную. Каждый из потоков в группе выполняет следующую за директивой команду (или блок команд, определённый в {}-скобках). После выполнения, потоки «сливаются» в один.

```
#pragma omp parallel {

    // Код внутри блока выполняется параллельно.
    // Будет написано несколько раз Hello!

    printf("Hello!\n");

  }
```

`for`

Директива for разделяет цикл for между текущей группой потоков, так что каждый поток в группе обрабатывает свою часть цикла.

```
#pragma omp for

 for(int n=0; n<10; ++n)
```

```
 {
   printf(" %d", n);
 }
 printf("\n");
```

parallel и for можно комбинировать:

```
#pragma omp parallel for

 for(int n=0; n<10; ++n) printf(" %d", n);

 printf("\n");
```

num_threads

Чтобы задать количество потоков в группе, можно воспользоваться параметром num_threads:

```
#pragma omp parallel num_threads(3)

 {
   // Данный код будет выполнен тремя потоками.
   // Части цикла будут поделены между
   // тремя потоками текущей группы потоков.

   #pragma omp for
   for(int n=0; n<10; ++n) printf(" %d", n);
 }
```

scheduling

Программист может контролировать то, каким образом потоки будут загружаться работой при обработке цикла. Существует несколько вариантов.

В каждом из вариантов можно указать размер блока итераций выполняемых подряд одним потоком.

static

Вариант по умолчанию. Итерации цикла распределены между потоками еще до запуска программы

```
#pragma omp for schedule(static, 4)
```

dynamic

Итерации цикла распределяются динамически по ходу выполнения программы.

```
#pragma omp for schedule(dynamic, 4)
```

guided

То же, что и dynamic, только размер блока уменьшается экспоненциально.

ordering

Заставить OpenMP выполнять итерации цикла по порядку

```
#pragma omp for ordered schedule(dynamic)
 for(int n=0; n<100; ++n)
 {
   files[n].compress();
   #pragma omp ordered
   send(files[n]);
 }
```

critical

Данная директива указывает OpenMP, что следующий блок кода является критической секцией, т.е. только один поток может приступить к исполнению этого блока. Если другой поток доходит до этого

блока, то он блокируется, т.е. ждет когда поток, вошедший в критическую секцию, выйдет из нее. Также у секций может быть имя. Если какой-то поток вошел в секцию section_x, то все другие блоки с таким же именем становятся заблокированными.

```
#pragma omp critical section_x
{
      X+=5;
}

#pragma omp critical
{
      cout << "Hello from thread" << omp_get_thread_num()<<endl;
}
```

Функции

Для работы с функциями необходимо подключить заголовочный файл `omp.h`

```
#include <omp.h>
```

`omp_set_num_threads(int num_threads)`

Устанавливает кол-во потоков, которые будут использоваться в следующем распараллеленном блоке.

`omp_get_num_threads(void)`

Кол-во потоков, исполняющихся в данный момент.

`int omp_get_thread_num(void)`

Возвращает номер текущего потока.

`void omp_set_nested(int nested)`

Включает или выключает вложенный параллелизм. По умолчанию выключен.

`double omp_get_wtime(void)`

Таймер, в секундах.

Summary

- **Группа потоков** – те потоки, которые выполняются в данный момент.
- **При запуске программы**, группа содержит **один поток**.
- Директива **parallel** разделяет текущий поток в новую группу потоков, пока не будет достигнут конец следующего выражения/блока выражений, затем группа потоков сливается в один поток.
- **for** разделяет цикл for на части, и отдаёт каждую часть потоку из текущей группы. Данная директива **не создаёт новых потоков**, она всего лишь делит работ между потоками текущей группы.
- **parallel for** - краткая запись двух команд: parallel и for. Parallel создаёт новую группу потоков, затем for раздает каждому потоку из этой группы часть цикла.

Если программа **не содержит** директивы parallel, то она выполняется **единственным потоком**.

Источники

https://computing.llnl.gov/tutorials/openMP/
https://software.intel.com/ru-ru/blogs/2011/11/21/openmp-c

12. Основные приёмы и способы разработки параллельных программ с применением технологии MPI.

Message Passing Interface (MPI, интерфейс передачи сообщений) — программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. Разработан Уильямом Гроуппом, Эвином Ласком (англ.) и другими. MPI использует модель с распределённой памятью.

В вычислительных системах с распределенной памятью процессы работают независимо друг от друга и имеют собственную память.

В рамках MPI для решения задачи разрабатывается одна программа, она запускается на выполнение одновременно на всех имеющихся процессорах.

- Для организации различных вычислений на разных процессорах:
  - Есть возможность подставлять разные данные для программы на разных процессорах;
  - Имеются средства для идентификации процессора, на котором выполняется программа.
- Такой способ организации параллельных вычислений обычно именуется как модель "одна программа множество процессов" (single program multiple processes or SPMP)

MPI - это стандарт, которому должны удовлетворять средства организации передачи сообщений. MPI – это программные средства, которые обеспечивают возможность передачи сообщений и при этом соответствуют всем требованиям стандарта MPI:

- программные средства должны быть организованы в виде библиотек программных модулей (библиотеки MPI);
- должны быть доступны для наиболее широко используемых алгоритмических языков C и Fortran.

Под параллельной программой в рамках MPI понимается множество одновременно выполняемых процессов, процессы могут выполняться на разных процессорах; вместе с этим, на одном процессоре могут располагаться несколько процессов. Каждый процесс параллельной программы порождается на основе копии одного и того же программного кода (модель SPMP).

Исходный программный код разрабатывается на алгоритмических языках C, C++ или Fortran с использованием библиотеки MPI. Количество процессов и число используемых процессоров определяется в момент запуска параллельной программы средствами среды исполнения MPI программ. Все процессы программы последовательно перенумерованы. Номер процесса именуется рангом процесса.

Коммуникаторы
Коммуникатор в MPI - специально создаваемый служебный объект, объединяющий в своем составе группу процессов и ряд дополнительных параметров (контекст)
В ходе вычислений могут создаваться новые и удаляться существующие коммуникаторы. Один и тот же процесс может принадлежать разным коммуникаторам. Все имеющиеся в параллельной программе процессы входят в состав создаваемого по умолчанию коммуникатора с идентификатором MPI_COMM_WORLD. При необходимости передачи данных между процессами из разных групп необходимо создавать глобальный коммуникатор (intercommunicator).

Создание коммуникатора:
int MPI_comm_create (MPI_Comm oldcom, MPI_Group group, MPI_Comm *newcomm);

Дублирование коммуникатора:
int MPI_Comm_dup (MPI_Comm oldcom, MPI_comm *newcomm);

Удаление коммуникатора:
int MPI_Comm_free (MPI_Comm *comm);

Интеробмены - обмены между коммуникаторами
При выполнении интеробмена процессу-источнику сообщения указывается ранг адресата относительно удаленной группы, а процессу- получателю — ранг источника (также относительно удаленной по отношению к получателю группы). Обмен выполняется между лидерами обеих групп (MPI-1). Предполагается, что в обеих группах есть, по крайней мере, по одному процессу, который может обмениваться сообщениями со своим партнером.

Интеробмен возможен, только если создан соответствующий интеркоммуникатор, а это можно сделать с помощью подпрограммы:

int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *new_intercomm)

Параметры:

local_comm — локальный интракоммуникатор;

local_leader — ранг лидера в локальном коммуникаторе (обычно 0);

peer_comm — удаленный коммуникатор;

remote_leader — ранг лидера в удаленном коммуникаторе (обычно 0);

tag — тег интеркоммуникатора, используемый лидерами обеих групп для обменов в контексте родительского коммуникатора.

Выходной параметр — интеркоммуникатор (new_intercomm). «Джокеры» в качестве параметров использовать нельзя. Вызов этой подпрограммы должен выполняться в обеих группах процессов, которые должны быть связаны между собой. В каждом из этих вызовов используется локальный интракоммуникатор, соответствующий данной группе процессов.

Передача сообщений
Основу MPI составляют операции передачи сообщений. Среди предусмотренных в составе MPI функций различаются:
– парные (point-to-point) операции между двумя процессами;
– коллективные (collective) коммуникационные действия для одновременного взаимодействия нескольких процессов.
– парные операции передачи данных выполняются для процессов, принадлежащих одному и тому же коммуникатору,
– Коллективные операции применяются одновременно для всех процессов коммуникатора.
Указание используемого коммуникатора является обязательным для операций передачи данных в MPI.

Типы данных
При выполнении операций передачи сообщений для указания передаваемых или получаемых данных в функциях MPI необходимо указывать тип пересылаемых данных. MPI содержит большой набор базовых типов данных, во многом совпадающих с типами данных в алгоритмических языках C и Fortran. В MPI имеются возможности для создания новых производных типов данных для более точного и краткого описания содержимого пересылаемых сообщений.
Соответствие типов данных MPI и C

Тип MPI	Тип C
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	Double
MPI_FLOAT	Float
MPI_INT	Int
MPI_LONG	Long

MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Виртуальные топологии

Логическая топология линий связи между процессами имеет структуру полного графа (независимо от наличия реальных физических каналов связи между процессорами). В MPI имеется возможность представления множества процессов в виде решетки произвольной размерности. При этом, граничные процессы решеток могут быть объявлены соседними и, тем самым, на основе решеток могут быть определены структуры типа тор. В MPI имеются средства и для формирования логических (виртуальных) топологий любого требуемого типа.

Декартовы топологии (решетки)

В декартовых топологиях множество процессов представляется в виде прямоугольной решетки, а для указания процессов используется декартова системы координат. Для создания декартовой топологии (решетки) в MPI предназначена функция:

int MPI_Cart_create(MPI_Comm oldcomm, int ndims, int *dims, int *periods, int reorder, MPI_Comm *cartcomm), где:

- oldcomm - исходный коммуникатор,
- ndims - размерность декартовой решетки,
- dims - массив длины ndims, задает количество процессов в каждом измерении решетки,
- periods - массив длины ndims, определяет, является ли решетка периодической вдоль каждого измерения,
- reorder - параметр допустимости изменения нумерации процессов,
- cartcomm – создаваемый коммуникатор с декартовой топологией процессов.

Для определения декартовых координат процесса по его рангу можно воспользоваться функцией:
int MPI_Card_coords(MPI_Comm comm,int rank,int ndims,int *coords)

Обратное действие – определение ранга процесса по его декартовым координатам – обеспечивается при помощи функции:
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)

Топология графа

Для создания топологии с графом данного вида необходимо выполнить следующий программный код:

```
int index[] = { 4,1,1,1,1 };
int edges[] = { 1,2,3,4,0,0,0,0 };
MPI_Comm StarComm;
MPI_Graph_create(MPI_COMM_WORLD, 5, index, edges, 1, &StarComm);
```

Получение списка соседей:
int MPI_Graph_neighbors_count(MPI_Comm comm,int rank, int *nneighbors);

Структура MPI приложения

Инициализация и завершение MPI программ Первой вызываемой функцией MPI должна быть функция: int MPI_Init (int *agrc, char ***argv), служащая для инициализации среды выполнения MPI программы; параметрами функции являются количество аргументов в командной строке и текст самой командной строки.)

Последней вызываемой функцией MPI обязательно должна являться функция: int MPI_Finalize (void).
Определение количества процессов в выполняемой параллельной программе осуществляется при помощи функции:
int MPI_Comm_size (MPI_Comm comm, int *size).
Для определения ранга процесса используется функция:
int MPI_Comm_rank (MPI_Comm comm, int *rank).

Size и rank, переменные, куда будет сохранено значение размера (количества процессов) и ранга процесса в коммуникаторе Comm.

Простейшая программа

```
#include "mpi.h"
int main ( int argc, char *argv[] )
{
	int ProcNum, ProcRank;
	MPI_Init ( &agrc, &argv );
	MPI_Comm_size ( MPI_COMM_WORLD, &ProcNum);
	MPI_Comm_rank ( MPI_COMM_WORLD, &ProcRank);
	MPI_Finalize();
	return 0;
}
```

Коммуникатор MPI_COMM_WORLD создается по умолчанию и представляет все процессы выполняемой параллельной программы. Ранг, получаемый при помощи функции MPI_Comm_rank, является рангом процесса, выполнившего вызов этой функции, и, тем самым, переменная ProcRank будет принимать различные значения в разных процессах.

Передача сообщений

Для передачи сообщения процесс-отправитель должен выполнить функцию:
	int MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm), где
buf – адрес буфера памяти, в котором располагаются данные отправляемого сообщения,
count – количество элементов данных в сообщении,
type - тип элементов данных пересылаемого сообщения,
dest - ранг процесса, которому отправляется сообщение,
tag - значение-тег, используемое для идентификации сообщений,
comm - коммуникатор, в рамках которого выполняется передача данных

Отправляемое сообщение определяется через указание блока памяти (буфера), в котором это сообщение располагается. Используемая для указания буфера триада (buf, count, type) входит в состав параметров практически всех функций передачи данных.
Процессы, между которыми выполняется передача данных, обязательно должны принадлежать коммуникатору, указываемому в функции MPI_Send.
Параметр tag используется только при необходимости различения передаваемых сообщений, в противном случае в качестве значения параметра может быть использовано произвольное целое число.

Широковещательная рассылка данных может быть обеспечена при помощи функции MPI:
	int MPI_Bcast(void *buf,int count,MPI_Datatype type, int root,MPI_Comm comm), где
buf, count, type – буфер памяти с отправляемым сообщением (для процесса с рангом 0), и для приема сообщений для всех остальных процессов;
root - ранг процесса, выполняющего рассылку данных;
comm - коммуникатор, в рамках которого выполняется передача данных.

Функция MPI_Bcast осуществляет рассылку данных из буфера buf, содержащего count элементов типа type с процесса, имеющего номер root, всем процессам, входящим в коммуникатор comm.
Функция MPI_Bcast определяет коллективную операцию, вызов функции MPI_Bcast должен быть осуществлен всеми процессами указываемого коммуникатора.

Указываемый в функции MPI_Bcast буфер памяти имеет различное назначение в разных процессах:
– Для процесса с рангом root, с которого осуществляется рассылка данных, в этом буфере должно находиться рассылаемое сообщение;
– Для всех остальных процессов указываемый буфер предназначен для приема передаваемых данных.

Процедура сбора и последующего суммирования данных является примером часто выполняемой коллективной операции передачи данных от всех процессов одному процессу, в которой над собираемыми значениями осуществляется та или иная обработка данных. Для выполнения этой операции используется функция:

int MPI_Reduce(void *sendbuf, void *recvbuf,int count, MPI_Datatype type, MPI_Op op,int root,MPI_Comm comm), где
- sendbuf - буфер памяти с отправляемым сообщением;
- recvbuf – буфер памяти для результирующего сообщения (только для процесса с рангом root);
- count - количество элементов в сообщениях;
- type – тип элементов сообщений;
- op - операция, которая должна быть выполнена над данными, (MPI_MAX (максимум), MPI_MIN (минимум), MPI_SUM (сумма), MPI_PROD (произведение), MPI_BOR (побитовое или), MPI_LOR (логическое или));
- root - ранг процесса, на котором должен быть получен результат;
- comm - коммуникатор, в рамках которого выполняется операция.

Функция MPI_Reduce определяет коллективную операцию и, тем самым, вызов функции должен быть выполнен всеми процессами указываемого коммуникатора, все вызовы функции должны содержать одинаковые значения параметров count, type, op, root, comm. Передача сообщений должна быть выполнена всеми процессами, результат операции будет получен только процессом с рангом root. Выполнение операции редукции осуществляется над отдельными элементами передаваемых сообщений.

Рисунок 1 – схема выполнения функции reduce.

Приём сообщений

Для приема сообщения процесс-получатель должен выполнить функцию:

int MPI_Recv(void *buf, int count, MPI_Datatype type, int source,int tag, MPI_Comm comm, MPI_Status *status), где
buf, count, type – буфер памяти для приема сообщения;
source - ранг процесса, от которого должен быть выполнен прием сообщения;
tag - тег сообщения, которое должно быть принято для процесса;
comm - коммуникатор, в рамках которого выполняется передача данных;

status – указатель на структуру данных с информацией о результате выполнения операции приема данных.

Буфер памяти должен быть достаточным для приема сообщения, а тип элементов передаваемого и принимаемого сообщения должны совпадать; при нехватке памяти часть сообщения будет потеряна и в коде завершения функции будет зафиксирована ошибка переполнения. При необходимости приема сообщения от любого процесса- отправителя для параметра source может быть указано значение MPI_ANY_SOURCE (так называемые wildcards, джокеры). При необходимости приема сообщения с любым тегом для параметра tag может быть указано значение MPI_ANY_TAG.

Параметр status позволяет определить ряд характеристик принятого сообщения:
status.MPI_SOURCE – ранг процесса-отправителя принятого сообщения;
status.MPI_TAG - тег принятого сообщения.

Функция MPI_Get_count(MPI_Status *status, MPI_Datatype type, int *count) возвращает в переменной count количество элементов типа type в принятом сообщении.
Функция MPI_Recv является блокирующей для процесса-получателя, т.е. его выполнение приостанавливается до завершения работы функции. Таким образом, если по каким-то причинам ожидаемое для приема сообщение будет отсутствовать, выполнение параллельной программы будет блокировано.

Другие вариации функций передачи:

Название режима	Условие завершения	Функция
Стандартная передача	Как для синхронной или буферизованной	MPI_SEND
Синхронная передача	Завершается, когда завершен прием	MPI_SSEND
Буферизованная передача	Всегда завершается	MPI_BSEND
Передача по готовности	Всегда завершается	MPI_RSEND
Прием	Завершается, когда сообщение принято	MPI_RECV

Программист, использующий стандартный режим передачи, должен следовать следующим рекомендациям:

- Он не должен рассчитывать, что передача завершится перед началом приема сообщения. Если передача является блокирующей, то в некоторых случаях может возникнуть тупик.
- Не должен рассчитывать, что передача завершится после начала приема сообщения. В этом случае порядок приема последовательности сообщений может быть нарушен.
- Процессы должны принимать и обрабатывать все сообщения, адресованные им.

Синхронная передача может быть существенно медленнее стандартной. Однако она не приведет к перегрузке коммуникационной сети сообщениями и обеспечит детерминированное поведение программы. Использование этого режима передачи также облегчает отладку параллельного приложения.
Буферизованная передача гарантирует немедленное завершение, поскольку сообщение вначале копируется в системный буфер, а затем доставляется. Недостатком ее является необходимость выделения специальных буферов, потребляющих ресурсы системы.
Передача по готовности предполагает инициирование передачи в момент, когда приемник вызывает соответствующий ей прием. Этот режим гарантирует отсутствие в коммуникационной сети блуждающих сообщений.
Порядок приема сообщений заранее не определен и зависит от условий выполнения параллельной программы (более того, этот порядок может изменяться от запуска к запуску). Если это не приводит

к потере эффективности, следует обеспечивать однозначность расчетов и при использовании параллельных вычислений, указывая ранг процесса-отправителя или используя тэг.

Пример

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
	int world_size, world_rank, hello_number;
	MPI_Init(&argc, &argv);
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);
	MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
	if (world_rank == 0) {
		printf("Hello world from process %d\n", world_rank);
		MPI_Status status;
		for (int i = 1; i < world_size; i++) {
		MPI_Recv(&hello_number, 1, MPI_INT, i, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
		printf("Hello world from process %d\n", hello_number);
		}
		}
	else {
		MPI_Send(&world_rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
	}
MPI_Finalize(); }
```

SendRecv

В ситуациях, когда требуется выполнить взаимный обмен данными между процессами, безопаснее (так как не возникает deadlock (тупики)) использовать совмещенную операцию MPI_Sendrecv.

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
int dest, int sendtag, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int source, MPI_Datatypea recvtag,
MPI_Comm comm, MPI_Status *status)
```

Структура MPI приложения

Можно рекомендовать при увеличении объема разрабатываемых программ выносить программный код разных процессов в отдельные программные модули (функции). Общая схема MPI программы в этом случае будет иметь вид:

```
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
if ( ProcRank == 0 ) DoProcess0();
else if ( ProcRank == 1 ) DoProcess1();
else if ( ProcRank == 2 ) DoProcess2();
```

Синхронизация процессов

Синхронизация процессов, т.е. одновременное достижение процессами тех или иных точек процесса вычислений, обеспечивается при помощи функции MPI:

```
int MPI_Barrier(MPI_Comm comm);
```

Функция MPI_Barrier определяет коллективную операцию, при использовании должна вызываться всеми процессами коммуникатора. Продолжение вычислений любого процесса произойдет только после выполнения функции MPI_Barrier всеми процессами коммуникатора.

Группы процессов

Процессы параллельной программы объединяются в группы. В группу могут входить все процессы параллельной программы или в группе может находиться только часть имеющихся процессов. Один и тот же процесс может принадлежать нескольким группам. Управление группами процессов предпринимается для создания на их основе коммуникаторов. Группы процессов могут быть

созданы только из уже существующих групп. В качестве исходной группы может быть использована группа, связанная с предопределенным коммуникатором MPI_COMM_WORLD:

int MPI_Comm_group (MPI_Comm comm, MPI_Group *group)

На основе существующих групп, могут быть созданы новые группы – создание новой группы newgroup из существующей группы oldgroup, которая будет включать в себя n процессов, ранги которых перечисляются в массиве ranks:

int MPI_Group_incl(MPI_Group oldgroup,int n, int *ranks, MPI_Group *newgroup);

создание новой группы newgroup из группы oldgroup, которая будет включать в себя n процессов, ранги которых не совпадают с рангами, перечисленными в массиве ranks:

int MPI_Group_excl(MPI_Group oldgroup,int n, int *ranks, MPI_Group *newgroup);

Также существуют функции для объединения, пересечения и разности групп.

int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup);

int MPI_Group_intersection (MPI_Group group1, MPI_Group group2, MPI_Group *newgroup);

int MPI_Group_difference (MPI_Group group1, MPI_Group group2, MPI_Group *newgroup);

Получение информации о группе процессов: – получение количества процессов в группе: – получение ранга текущего процесса в группе:

int MPI_Group_size (MPI_Group group, int *size);

После завершения использования группа должна быть удалена:

int MPI_Group_rank (MPI_Group group, int *rank);

Процесс может входить в несколько групп. Подпрограмма MPI_Group_translate_ranks преобразует ранг процесса в одной группе в его ранг в другой группе:

int MPI_Group_translate_ranks(MPI_Group group1, int n, int *ranks1, MPI_Group group2, int *ranks2).

Для контроля правильности выполнения все функции MPI возвращают в качестве своего значения код завершения. При успешном выполнении функции возвращаемый код равен MPI_SUCCESS. Другие значения кода завершения свидетельствуют об обнаружении тех или иных ошибочных ситуаций в ходе выполнения функций:

- MPI_ERR_BUFFER – неправильный указатель на буфер;
- MPI_ERR_COMM – неправильный коммуникатор;
- MPI_ERR_RANK – неправильный ранг процесса и др.

Подсчёт времени

Получение времени текущего момента выполнения программы обеспечивается при помощи функции: double MPI_Wtime(void).

Точность измерения времени может зависеть от среды выполнения параллельной программы. Для определения текущего значения точности может быть использована функция: double MPI_Wtick(void) (время в секундах между двумя последовательными показателями времени аппаратного таймера используемой системы).

13. Основные приёмы и способы разработки параллельных программ с применением технологии MPI.

PVM (параллельная виртуальная машина) - это побочный продукт продвижения гетерогенных сетевых исследовательских проектов, распространяемый авторами и институтами, в которых они работают. Общими целями этого проекта являются исследование проблематики и разработка решений в области гетерогенных параллельных вычислений. PVM представляет собой набор программных средств и библиотек, которые эмулируют общецелевые, гибкие гетерогенные вычислительные структуры для параллелизма во взаимосвязанных компьютерах с различными архитектурами. Главной же целью системы PVM является обеспечение возможности совместного использования группы компьютеров совместно для взаимосвязанных или параллельных вычислений.

PVM – это некоторая альтернатива MPI, но менее распространенная.

PVM можно использовать программируя на различных языках (например, C/C++, Fortran, Perl, Java, Matlab).

Гетерогенность выражается разнородностью:

1. по формату данных на узлах.
2. по быстродействию узлов.
3. по архитектуре узлов.
4. по загрузке узлов.
5. по загрузке сети.

Преимущества PVM:

- Сравнительно низкая стоимость.
- Возможность повышения производительности за счет динамического распределения задач по узлам.
- Устойчивость в процессе решения задач (интерактивное управление).
- Наращивание и модификация программ.

Основные постулаты, взятые за основу для PVM:

1. Конфигурируемый пользователем пул хостов: вычислительные задачи приложения выполняются с привлечением набора машин, которые выбираются пользователем для данной программы PVM. Как однопроцессорные машины, так и аппаратное обеспечение мультипроцессоров (включая компьютеры с разделяемой и распределенной памятью) могут быть составной частью пула хостов. Пул хостов

может изменяться добавлением и удалением машин в процессе работы (важная возможность для поддержания минимального уровня ошибок).

2. Прозрачность доступа к оборудованию: прикладные программы могут ``видеть'' аппаратную среду как группу виртуальных вычислительных элементов без атрибутов или эксплуатировать по выбору возможности специфических машин из пула хостов путем ``перемещения'' определенных счетных задач на наиболее подходящие для их решения компьютеры.
3. Вычисления, производимые с помощью процессов: единицей параллелизма в PVM является задача (часто, но не всегда совпадает с процессом в системе UNIX) - независимый последовательный поток управления, который может быть либо коммуникационным, либо вычислительным. PVM не содержит и не навязывает карты связей процессов; характерно, что составные задачи могут выполняться на одном процессоре.
4. Модель явного обмена сообщениями: группы вычислительных задач, каждая из которых выполняет часть ``нагрузки'' приложения - используется декомпозиция по данным, функциям или гибридная, - взаимодействуют, явно посылая сообщения друг другу и принимая их. Длина сообщения ограничена только размером доступной памяти.
5. Поддержка гетерогенности: система PVM поддерживает гетерогенность системы машин, сетей и приложений. В отношении механизма обмена сообщениями PVM допускает сообщения, содержащие данные более одного типа, для обмена между машинами с различным представлением данных.
6. Поддержка мультипроцессоров: PVM использует оригинальные возможности обмена сообщениями для мультипроцессоров с целью извлечения выгоды от использования базового оборудования. Производители часто поддерживают собственные, оптимизированные для своих систем PVM, которые становятся коммуникационными в их общей версии.

Работа PVM

1. Создание параллельной виртуальной машины. На каждом хосте при этом запускается демон (pvmd). Каждому процессу присваивается идентификатор TD (аналогично рангу в MPI, но на системном уровне есть отличия).

   Идентификатор – это ссылка на служебную информацию с описание процесса. TD назначается локальным демоном. Доступ к TD осуществляется с помощью библиотечной функции.

2. Каждому компьютеру присваивается имя, связанное с его архитектурой. Каждый процесс может передавать сообщения другому процессу. Так же, можно создавать динамические группы (эквивалент коммуникаторам).
3. При запуске демона в каталоге /tmp (служебный каталог временного хранения) создается файл pvmd.UTD, расширение которого совпадает с числовым идентификатором пользователя. Этот файл является блокирующим – если демон не работает, то такой хост нельзя добавить в систему виртуальной машины. Чтобы удалить pvd.UID, необходимо с помощью консоли ввести команду halt (остановить всё).
4. Демон параллельной виртуальной машины – это гибкое средство, позволяющее на одном компьютере работать нескольким пользователям PVM. Причем, для каждого пользователя запускается свой демон, и они работают независимо.

Демон играет роль маршрутизатора сообщений. Он управляет процессами, обеспечивает продолжение работы при аварийном завершении. pvmd, запущенный вручную - называется главным (master). Остальные называются исполнителями (slave). Главный демон запускает остальные slave.

Модель передачи сообщений

Сообщениями обмениваются задачи под управлением демонов. При обмене данные преобразуются в формат XDR, т.о. преодолевается гетерогенность. При отправке сообщения отправитель не блокируется.

Сообщение передается в коммуникационную сеть, откуда попадает в буфер приема (адресата).

- Буферы обмены выделяются динамически.
- Последовательность сообщений от одного отправителя сохраняются при приеме.
-

Пример простейшей программы

```
// программа для master
#include "pvm3.h"
main()
{
        int err, tid, msgtag;
        char buf[100];
        printf("master:tid t%i\n", pvm_mytid());
        err= pvm_spawn("f1", (char**)0, 0, "",1, &tid);
        if (err == 1) {
        msgtag = 1;
        pvm_recv(tid, msgtag);
```

```
        pvm_upkstr(buf);
        printf("Получено сообщение от tid= %i: %s\n", tid, buf);
                        }
        else printf("Ошибка\n");
        pvm_exit();
}
```

Лекция 12

Структура программы PVM.

f1

```
#include “pvm3.h”
main ()
{
int ptid, msgtag;
char buf [100];
ptid=pvm-parent (); // определяется идентификатор базового процесса
strcopy (buf, “Задача f1”); // занесение в буфер
msgtag=1;
pvm-initsend ();
PvmDataDefault (); //режимы инициирования по умолчанию
pvm_pkstr (buf);
pvm_send (ptid, msgtag);
pvm_exit ();
}
```

Выводы по PVM:

1. Если платформа в виде многопроцессорного компьютера, то рекомендуется использовать технологию MPI;
2. Если предполагается использовать различные по настройкам режимы обмена, то используем MPI;
3. Если используется гетерогенный кластер (разнородный), то используем PVM;
4. Динамическое распределение задач, то используем PVM.

14. Классические алгоритмы синхронизации

Проблемы синхронизации.

Синхронизация отдельных потоков является одной из основных задач. Отсутствие синхронизации может привести не только к неправильному выполнению программы, но и краху системы в целом.

Взаимное исключение – попытка получения доступа к одной переменной из нескольких потоков.

Случай тупика (deadlock). Тупик возникает в случае, если в ходе захвата ресурса, каждый из потоков блокирует захваченный ресурс. Решение:

1. Не допускать тупики
2. Предотвращать тупики (не допускать блокировки одним потоком более одного ресурса)
3. Сброс и восстановление
4. Игнорирование (win рулит)

Критическая секция – часть программы, результат выполнения которой может непредсказуемо меняться, если переменные, относящиеся к ней, изменяются другими потоками в то время, когда выполнение этой части еще не завершено.

Механизмы синхронизации.

Синхронизация требуется в случае:

- Когда несколько потоков (процессов) используют общий ресурс, в этом случае взаимоисключения должны осуществляться в принудительном порядке
- Когда несколько потоков (процессов) используют один или несколько ресурсов, тогда процесс, завершающий работу не должен влиять на работу других процессов
- Когда процесс находится в режиме постоянного ожидания доступа к критическому ресурсу
- Когда к критическому ресурсу не имеет доступ ни один процесс, каждый следующий процесс должен получать его незамедлительно
- Работа процессов не должна учитывать относительную скорость работы самого процесса, так и других;
- В любом случае ресурс должен захватывать процессом только на ограниченное время.

Сформулируем пять условий, которые должны выполняться для хорошего программного алгоритма организации взаимодействия процессов, имеющих критические участки, если они могут проходить их в произвольном порядке:
1. Задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд взаимоисключения. При этом предполагается, что основные инструкции языка программирования являются атомарными операциями.
2. Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.
3. Если процесс Pi исполняется в своем критическом участке, то не существует никаких других процессов, которые исполняются в своих соответствующих критических секциях. Это условие получило название условия взаимоисключения (mutual exclusion).
4. Процессы, которые находятся вне своих критических участков и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные критические участки. Если нет процессов в критических секциях, и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в remainder section, должны принимать решение о том, какой процесс войдет в свою критическую секцию. Такое решение не должно приниматься бесконечно долго. Это условие получило название условия прогресса (progress).

5. Не должно возникать бесконечного ожидания для входа процесса в свой критический участок. От того момента, когда процесс запросил разрешение на вход в критическую секцию, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои критические участки лишь ограниченное число раз. Это условие получило название условия ограниченного ожидания (bound waiting).

Надо заметить, что описание соответствующего алгоритма означает описание способа организации пролога и эпилога для **критической секции.**

Алгоритмы синхронизации
Запрет прерываний

Наиболее простым решением поставленной задачи является следующая организация пролога и эпилога:

```
while (условие) {
запретить все прерывания
критическая секция
разрешить все прерывания
остальной код
}
```

Поскольку, выход процесса из состояния исполнения без его завершения осуществляется по прерыванию, то внутри критической секции никто не может вмешаться в его работу. Однако такое решение чревато далеко идущими последствиями, поскольку разрешает процессу пользователя разрешать и запрещать прерывания во всей вычислительной системе. Допустим, что в результате ошибки или злого умысла пользователь запретил прерывания в системе и зациклил или завершил свой процесс. Без перезагрузки системы в такой ситуации не обойтись.

Переменная замок. Возьмем некоторую переменную, доступную всем процессам, и положим ее начальное значение равным 0. Процесс может войти в критическую секцию только тогда, когда значение этой переменной-замка равно 0, одновременно изменяя ее значение на 1 -- закрывая замок. При выходе из критической секции процесс сбрасывает ее значение в 0 -- замок открывается.

```
shared int lock = 0;
while (условие) {
while(lock); lock = 1;
критическая секция
lock = 0;
остальной код
}
```

К сожалению, внимательное изучение показывает, что такое решение, не удовлетворяет условию взаимоисключения, так как действие while(lock); lock = 1; не является атомарным. Допустим, что процесс P0 протестировал значение переменной lock и принял решение двигаться дальше. В этот момент, еще до присваивания переменной lock значения 1, планировщик передал управление процессу P1. Он тоже изучает содержимое переменной lock и тоже принимает решение войти в критический участок. Мы получаем два процесса одновременно выполняющих свои критические секции.

Строгое чередование

Попробуем решить задачу сначала для двух процессов. Очередной подход будет также использовать общую для них обоих переменную с начальным значением 0. Только теперь она

будет играть не роль замка для *критического участка*, а явно указывать, кто может следующим войти в него. Для i-го процесса это выглядит так:

```
shared int turn = 0;

while (some condition) {
while(turn != i);
critical section
turn = 1-i;
remainder section
}
```

Очевидно, что *взаимоисключение* гарантируется, процессы входят в *критическую секцию* строго по очереди: P_0, P_1, P_0, P_1, P_0, ... Но наш алгоритм не удовлетворяет *условию прогресса*. Например, если значение turn равно 1, и процесс P_0 готов войти в *критический участок*, он не может сделать этого, даже если процесс P_1 находится в remainder section.

Флаги готовности

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени. Давайте попробуем исправить эту ситуацию. Пусть два наших процесса имеют разделяемый массив флагов готовности входа процессов в *критический участок*

```
shared int ready[2] = {0, 0};
```

Когда i-й процесс готов войти в *критическую секцию*, он присваивает элементу массива ready[i] значение равное 1. После выхода из *критической секции* он, естественно, сбрасывает это значение в 0. Процесс не входит в *критическую секцию*, если другой процесс уже готов к входу в *критическую секцию* или находится в ней.

```
while (some condition) {
  ready[i] = 1;
  while(ready[1-i]);
    critical section
  ready[i] = 0;
    remainder section
}
```

Полученный алгоритм обеспечивает *взаимоисключение*, позволяет процессу, готовому к входу в *критический участок*, войти в него сразу после завершения эпилога в другом процессе, но все равно нарушает *условие прогресса*. Пусть процессы практически одновременно подошли к выполнению пролога. После выполнения присваивания ready[0]=1 планировщик передал процессор от процесса 0 процессу 1, который также выполнил присваивание ready[1]=1. После этого оба процесса бесконечно долго ждут друг друга на входе в *критическую секцию*. Возникает ситуация, которую принято называть тупиковой (*deadlock*).

Команда «проверка и установка» (TestSet). Данная команда позволяет захватить общий ресурс, используя неделимую операцию проверки возможности захвата и захвата ресурса. В этом случае, попадая в блок, использующий критический ресурс, процесс обязан переустановить переменную, связанную с этим ресурсом с целью возможности организации доступа к нему другим процессам. В случае использования этой команды также будет использоваться активное ожидание изменения значения внутренней переменной. Захват ресурса новым процессом осуществляется внутри блока «проверки и установа».

Алгоритм Деккера. Предполагает, что в каждый момент времени ресурс захватывается только одним процессом, причём с каждым из работающих процессов связано собственное значение барьерной переменной. Эту переменную называют глобальной переменной, которая должна быть доступна всем процессам. Всякий процесс, желающий получить доступ к ресурсу проверяет разрешения доступа для себя.

Если два процесса пытаются перейти в критическую секцию одновременно, алгоритм позволит это только одному из них, основываясь на том, чья в этот момент очередь. Если один процесс уже вошёл в критическую секцию, другой будет ждать, пока первый покинет её. Это реализуется при помощи использования двух флагов (индикаторов "намерения" войти в критическую секцию) и переменной turn (показывающей, очередь какого из процессов наступила).
Процессы объявляют о намерении войти в критическую секцию; это проверяется внешним циклом «while». Если другой процесс не заявил о таком намерении, в критическую секцию можно безопасно войти (вне зависимости от того, чья сейчас очередь). Взаимное исключение всё равно будет гарантировано, так как ни один из процессов не может войти в критическую секцию до установки этого флага (подразумевается, что, по крайней мере, один процесс войдёт в цикл «while»). Это также гарантирует продвижение, так как не будет ожидания процесса, оставившего «намерение» войти в критическую секцию. В ином случае, если переменная другого процесса была установлена, входят в цикл «while» и переменная turn будет показывать, кому разрешено войти в критическую секцию. Процесс, чья очередь не наступила, оставляет намерение войти в критическую секцию до тех пор, пока не придёт его очередь (внутренний цикл «while»). Процесс, чья очередь пришла, выйдет из цикла «while» и войдёт в критическую секцию.

+ не требует специальных Test-and-set инструкций, по этому легко переносим на разные языки программирования и архитектуры компьютеров
-Действует только для двух процессов

Алгоритм Петерсона — программный алгоритм взаимного исключения потоков исполнения кода.
Перед тем как начать исполнение критической секции кода (то есть кода, обращающегося к защищаемым совместно используемым ресурсам), поток должен вызвать специальную процедуру (назовем ее EnterRegion) со своим номером в качестве параметра. Она должна организовать ожидание потока своей очереди входа в критическую секцию. После исполнения критической секции и выхода из нее, поток вызывает другую процедуру (назовем ее LeaveRegion), после чего уже другие потоки смогут войти в критическую область. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из предложений всегда следует после другого.
Тем самым работу в критическом участке продолжит процесс, которому было сделано последнее предложение.

- Как и алгоритм Деккера, действует только для 2 процессов
+ Более простая реализация, чем у алгоритма Деккера

Алгоритм булочной (Bakery algorithm)
Алгоритм Петерсона дает нам решение задачи корректной организации взаимодействия двух процессов. Давайте рассмотрим теперь соответствующий алгоритм для n взаимодействующих процессов, который получил название алгоритм булочной. Основная его идея выглядит так. Каждый вновь прибывающий клиент (он же процесс) получает талончик на обслуживание с номером. Клиент с наименьшим номером на талончике обслуживается следующим.
К сожалению, из-за неатомарности операции вычисления следующего номера алгоритм булочной не гарантирует, что у всех процессов будут талончики с разными номерами. В случае равенства номеров на талончиках у двух или более клиентов первым обслуживается клиент с меньшим значением имени (имена можно сравнивать в лексикографическом порядке). Разделяемые структуры данных для алгоритма -- это два массива
shared enum {false, true} choosing[n];
shared int number[n];
Изначально элементы этих массивов инициируются значениями false и 0 соответственно. Введем следующие обозначения

(a, b) < (c, d), если a < c или если a == c и b < d
max (a0, a1, …, an) -- это число k такое, что k >= ai для всех i = 0, …, n

Структура процесса Pi для алгоритма булочной приведена ниже

```
while (условие) {
choosing[i] = true;
number[i] = max (number[0], …, number[n-1]) + 1;
choosing[i] = false;
for (j = 0; j < n; j++){
while (choosing[j]);
while (number[j] ≠ 0 & & (number[j], j) < (number[i], i));
}
критическая секция
number[i] = 0;
остальной код
}
```

Специальные механизмы синхронизации – семафоры Дейкстры, мониторы Хора, очереди сообщений.

Мониторы

В 1974 году Хором (Hoare) был предложен механизм синхронизации высокого уровня, получивший название мониторов.

Мониторы представляют собой тип данных, который может быть с успехом внедрен в объектно-ориентированные языки программирования. Монитор обладает своими собственными переменными, определяющими его состояние. Значения этих переменных извне монитора могут быть изменены только с помощью вызова функций-методов, принадлежащих монитору. В свою очередь, эти функции-методы могут использовать в своей работе только данные, находящиеся внутри монитора и свои параметры.

Монитор состоит из:

- набора процедур, взаимодействующих с общим ресурсом
- мьютекса
- переменных, связанных с этим ресурсом
- инварианта, который определяет условия, позволяющие избежать состояние гонки

На абстрактном уровне можно описать структуру монитора следующим образом:

```
monitor monitor_name {
описание переменных ;
void mn(...){... }
{блок инициализации внутрениих переменных ;}
}
```

Здесь функции m1,..., mn представляют собой функции-методы монитора, а блок инициализации внутренних переменных содержит операции, которые выполняются только один раз: при создании монитора или при самом первом вызове какой-либо функции-метода до ее исполнения.

Важной особенностью мониторов является то, что в любой момент времени только один процесс может быть активен, т. е. находиться в состоянии готовность или исполнение, внутри данного монитора. Поскольку мониторы представляют собой особые конструкции языка программирования, то компилятор может отличить вызов функции, принадлежащей монитору, от вызовов других функций и обработать его специальным образом, добавив к нему пролог и эпилог, реализующий взаимоисключение. Так как обязанность конструирования механизма взаимоисключений возложена на компилятор, а не на программиста, работа программиста при

использовании мониторов существенно упрощается, а вероятность появления ошибок становится меньше.

Однако одних только взаимоисключений не достаточно для того, чтобы в полном объеме реализовать решение задач, возникающих при взаимодействии процессов. Нужны еще и средства организации очередности процессов, подобно семафорам full и empty. Для этого в мониторах было введено понятие условных переменных (condition variables), над которыми можно совершать две операции **wait** и **signal**, до некоторой степени похожие на операции **P** и **V** над семафорами.
Если функция монитора не может выполняться дальше, пока не наступит некоторое событие, она выполняет операцию wait над какой-либо условной переменной. При этом процесс, выполнивший операцию wait, блокируется, становится неактивным, и другой процесс получает возможность войти в монитор.

Когда ожидаемое событие происходит, другой процесс внутри функции-метода совершает операцию signal над той же самой условной переменной. Это приводит к пробуждению ранее заблокированного процесса, и он становится активным. Если несколько процессов дожидались операции signal для этой переменной, то активным становится только один из них. Что нам нужно предпринять для того, чтобы у нас не оказалось двух процессов, разбудившего и пробужденного, одновременно активных внутри монитора? Хор предложил, чтобы пробужденный процесс подавлял исполнение разбудившего процесса, пока он сам не покинет монитор. Несколько позже Хансен (Hansen) предложил другой механизм: разбудивший процесс покидает монитор немедленно после исполнения операции signal.

Семафоры
В теории операционных систем семафор представляет собой неотрицательную целую переменную, над которой возможны два вида операций: **P** и **V**.

P-операция над семафором представляет собой попытку уменьшения значения семафора на **1**. Если перед выполнением **P**-операции значение семафора было больше **0**, то **P**-операция выполняется без задержек. Если перед выполнением **P**-операции значение семафора было **0**, то процесс, выполняющий **P**-операцию, переводится в состояние ожидания до тех пор, пока значение семафора не станет большим **0**.

V-операция над семафором представляет собой увеличение значения семафора на **1**. Если при этом имеются процессы, задержанные на выполнении **P**-операции на данном семафоре, один из этих процессов выходит из состояния ожидания и может выполнить свою **P**-операцию.
Семафоры, принимающие два значения (с возможными значениями **0** и **1**), называются двоичными. Считающие семафоры (семафоры со счетчиками) принимают целые неотрицательные значения, большие двух.

Операции **P** и **V** являются неделимыми. Неделимость операций означает, что в каждый момент времени только один процесс может выполнять операцию **P** или **V** над данным семафором. Неделимость операции также означает, что если несколько процессов задерживаются на **P**-операции, то только один из них может успешно завершить свою **P**-операцию, если значение семафора стало больше **0**, при этом никаких предположений не делается о том, какой это будет процесс.

С каждым семафором связывается список (очередь) процессов, ожидающих разрешения пройти семафор. Операционная система может выполнять три действия над процессами. Она может назначить для исполнения готовый процесс. Она может заблокировать исполняющийся процесс и поместить его в список, связанный с конкретным семафором. Она может деблокировать процесс, тем самым, переводя его в готовое к исполнению состояние и позволяя ему когда-нибудь возобновить исполнение. Находясь в списке заблокированных, ожидающий процесс не проверяет

семафор непрерывно, как в случае активного ожидания. Вместо него на процессоре может исполняться другой процесс.

Операции **P** и **V** выполняются операционной системой в ответ на запрос, выданный некоторым процессом и содержащий имя семафора в качестве параметра. По операциям **P** и **V** выполняются следующие действия:

***P(S): if* S=1**
***then* S=S-1** /*закрыть семафор*/
else БЛОКИРОВАТЬ обратившийся процесс по S
V(S): if список процессов, ожидающих S, не пуст
then ДЕБЛОКИРОВАТЬ процесс, ожидающий S
***else* S=1** /*открыть семафор*/

При этом не определяется, какой из нескольких ожидающих процессов будет деблокирован.
Для реализации взаимного исключения, например, предотвращения возможности одновременного изменения двумя или более процессами общих данных, создается двоичный семафор **S**. Начальное значение этого семафора устанавливается равным **1**. Критические секции кода (секции, которые могут одновременно выполняться только одним процессом) обрамляются операциями **P(S)**(в начале секции) и **V(S)** (в конце секции).

P(S)
критическая секция
V(S)

Процесс, входящий в критическую секцию, выполняет операцию **P(S)** и переводит семафор в **0**. Если в критической секции уже находится другой процесс, то значение семафора уже равно **0**. Тогда второй процесс, желающий войти в критическую секцию, блокируется своей **P**-операцией до тех пор, пока процесс, находящийся в критической секции сейчас, не выйдет из нее, выполнив на выходе операцию **V(S)**.

Если начальное значение семафора равно единице, то взаимное исключение действительно гарантировано, так как процесс может выполнить **P**-операцию до того, как другой выполнит **V**-операцию. Кроме того, процесс без необходимости не перекрывает входы внутрь своей критической секции. Вход задерживается только тогда, когда некоторый другой процесс уже находится внутри своей собственной критической секции. Процесс отменяет вход, только, если значение семафора равно **0**.

Таким образом, в каждый момент времени процесс, желающий получить доступ к разделяемой переменной или разделяемым ресурсам, должен сделать это посредством критической секции, защищенной семафором.

Очереди сообщений
Механизм очередей сообщений позволяет процессам и потокам обмениваться структурированными сообщениями. Один или несколько процессов независимым образом могут посылать сообщения процессу – приемнику.

Очередь сообщений представляет возможность использовать несколько дисциплин обработки сообщений (FIFO, LIFO, приоритетный доступ, произвольный доступ).
При чтении сообщения из очереди удаления сообщения из очереди не происходит, и сообщение может быть прочитано несколько раз.

В очереди присутствуют не сами сообщения, а их адреса в памяти и размер. Эта информация размещается системой в сегменте памяти, доступном для всех задач, общающихся с помощью данной очереди

Основные функции управления очередью:

- Создание новой очереди
- Открытие существующей очереди
- Чтение и удаление сообщений из очереди
- Чтение без последующего удаления
- Добавление сообщения в очередь
- Завершение использование очереди
- Удаление из очереди всех сообщений
- Определение числа элементов в очереди

15. Средства автоматического распараллеливания

Средства автоматизированного распараллеливания следует использовать на первом этапе разработки параллельной программы, если у нее уже существует работающий последовательный аналог. В этом случае на первом этапе разработки параллельной программы рекомендуется применить средства автоматизированного распараллеливания к исходной последовательной программе.

В настоящее время все основные компиляторы Fortran и C/C++, предназначенные для разработки параллельных программ с использованием OpenMP, имеют возможности автоматического распараллеливания. Кроме того, эти компиляторы допускают установку различных уровней автоматического распараллеливания, а также генерируют отчеты по результатам распараллеливания.

Основным объектом автоматизированного распараллеливания с помощью специальных настроек компиляторов являются циклы. При распараллеливании циклов компилятор, во-первых, выделяет независимые по данным петли циклов. Во-вторых, он оценивает приблизительный эффект от распараллеливания цикла. Если оценка этого эффекта соответствует заданному уровню, то компилятор определяет локальные переменные. И, наконец, только затем производится само распараллеливание цикла с помощью библиотек процессов. По результатам проведенной или не проведенной работы выдается сообщение в файл отчета о распараллеливании с объяснением причин.

Как правило, других возможностей автоматизированного распараллеливания большинство компиляторов не имеет. Все это в полной мере относится к компиляторам Fortran и C/C++ компании Intel и некоторым другим. Как видим, вышеперечисленные возможности автоматизированного распараллеливания сравнительно невелики.

Кроме компиляторов существуют и более продвинутые программные системы в области автоматизированного распараллеливания.

В качестве примера отметим программный продукт Bert77, разрабатываемый компанией Paralogic. Программа Bert77 автоматически распараллеливает Fortran-программы в средах PVM или MPI. Распараллеливание осуществляется с использованием механизма обмена сообщениями.

Рассмотрим возможности автоматического распараллеливания программ с применением современных компиляторов компании Intel. Отметим, что в компиляторах Intel реализована только одна принципиальная возможность распараллеливания: это создание многопоточных приложений распараллеливанием цикла с помощью библиотек процессов.

Далее рассмотрим подробнее настройки режима автоматического распараллеливания, имеющиеся в компиляторах Fortran и C/C++ компании Intel:
- parallel - эта настройка позволяет компилятору автоматически создавать многопоточные версии программ с безопасным распараллеливанием циклов (подчеркнем, что кроме циклов в этом режиме больше ничего не распараллеливается);
- par_report{0|1|2|3} - эта настройка позволяет создавать отчеты различного уровня - 0, 1, 2 или 3 (наиболее подробный отчет имеет уровень 3) по результатам автоматического распараллеливания. Рекомендуется тщательно анализировать такие отчеты, чтобы четко понять, какие еще места программы могли бы быть распараллелены и почему это не было сделано. Возможно, компилятору не хватило информации для принятия решения о распараллеливании такого участка программы. В этом случае это можно сделать вручную;
- par_threshold[n] - эта настройка передает компилятору целое число n в диапазоне от 0 до 100. n является оценкой эффективности распараллеливания циклов в процентах. Эту оценку компилятор производит самостоятельно.

Современные компиляторы Intel содержат следующие специальные настройки для создания параллельных программ с применением средств OpenMP для параллельных вычислительных систем с общей памятью:
- openmp - эта настройка позволяет компилятору автоматически создавать многопоточные версии программ с использованием директив OpenMP;
- openmp_profile - эта настройка добавляет в создаваемую программу средства профилирования программы для последующего анализа с помощью программы VTune Performance Analyzer;
- openmp_stubs - эта настройка позволяет компилировать OpenMP программы в последовательном режиме. При этом предложения OpenMP игнорируются, а библиотека OpenMP используется редактором связей в последовательном режиме;
- openmp_report{0|1|2} - эта настройка позволяет создавать отчеты различного уровня - 0, 1 или 2 (наиболее подробный отчет имеет уровень 2) по результатам автоматического распараллеливания с использованием OpenMP. Рекомендуется тщательно анализировать такие отчеты, чтобы четко понять, какие еще места программы могли бы быть распараллелены и почему это не было сделано. Возможно, компилятору не хватило информации для принятия решения о распараллеливании того или иного участка программы. В этом случае это можно сделать вручную.

Итак, вышеперечисленные настройки компиляторов позволяют создавать параллельные версии программ как просто с многопоточным распараллеливанием циклов, так и с помощью директив OpenMP. Возможно также сочетание обоих этих режимов распараллеливания.

Простейшая команда компиляции программы prog.c с помощью компиляторов Intel:
icc -openmp prog.c

Примеры программ для автоматического распараллеливания от Intel:
- KAP/Pro Toolset - набор программных средств для распараллеливания больших расчетных программ на параллельных вычислительных системах с общей памятью;
- KAI C++ - компилятор C/C++ с широкими возможностями оптимизации и распараллеливания;
- Visual KAP - программа автоматического распараллеливания Fortran-программ в режиме визуального диалога;
- Visual KAP для OpenMP - инструмент визуальной генерации программ с использованием OpenMP.

В новой версии компиляторов Intel, начиная с версии 9.1, реализовано расширение OpenMP, предназначенное для распараллеливания программ для вычислительных систем с распределенной памятью. Это расширение известно под названием Cluster OpenMP. В нем имеется возможность объявлять области данных доступными для всех узлов кластера.

Компиляторы Intel, начиная с версии 9.1.x.xx, получили следующие дополнительные настройки, предназначенные для создания параллельных программ для параллельных вычислительных систем с распределенной памятью (кластеров):
- cluster-openmp - эта настройка позволяет компилятору создавать многопоточные версии программ с использованием директив расширенной версии OpenMP - Cluster OpenMP;
- cluster-openmp-profile - эта настройка добавляет в создаваемую программу с применением Cluster OpenMP средства профилирования программы для последующего анализа с помощью программы VTune Performance Analyzer;
- [no-]clomp-sharable-propagation - эта настройка позволяет создать отчет со списком переменных, которые должны быть объявлены программистом общедоступными для всех узлов кластера;
- [no-]clomp-sharable-info - эта настройка позволяет создать отчет со списком переменных, которые были автоматически объявлены компилятором общедоступными для всех узлов кластера. Далее рассмотрим, как применить эти настройки для создания параллельных программ для вычислительных систем с распределенной памятью.

Ленточные алгоритмы умножения матриц

В данных алгоритмах матрицы разбиваются на непрерывные последовательности строк или столб-цов (*полосы*). В простейшем случае полосой может служить отдельная строка или столбец.

В рассматриваемых ниже алгоритмах каждый процесс используется для вычисления одной строки результирующего произведения матриц *AB*. В этом случае процесс должен иметь доступ к соответствую-щей строке матрицы *A* и всей матрице *B*. Поскольку одновременное хранение всей матрицы B во всех процессах параллельного приложения требует чрезмерных затрат памяти, вычисления организуются та-ким образом, чтобы в каждый момент времени процессы содержали лишь часть элементов матрицы *B* (один столбец или одну строку), а доступ к остальной части обеспечивался бы при помощи передачи со-общений.

При описании ленточных алгоритмов предполагается, что количество процессов *N* совпадает с по-рядком перемножаемых матриц *A* и *B*, а матрицы являются квадратными.

Ленточный алгоритм 1

Вначале производится рассылка в процесс ранга *K* элементов *K*-й строки матрицы *A* и элементов *K*-го столбца матрицы *B*.

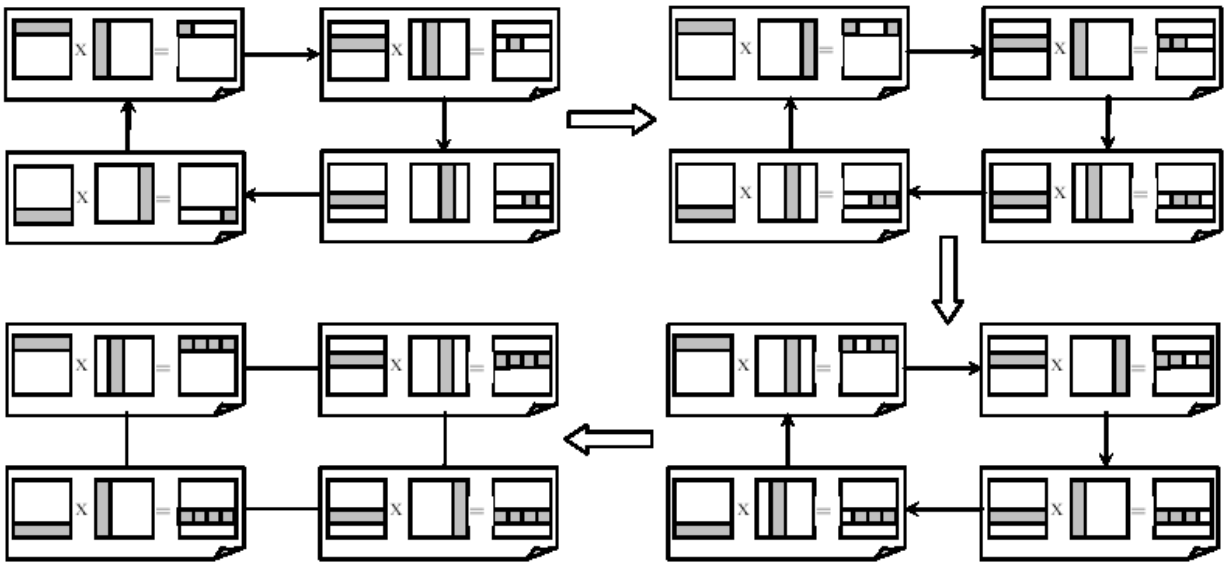
Затем запускается цикл (число итераций равно *N*), в ходе которого выполняются два действия:

1) выполняется перемножение строки матрицы *A* и столбца матрицы *B*, содержащихся в данном про-цессе, и результат записывается в соответствующий элемент строки *c*;

2) выполняется циклическая пересылка столбцов матрицы *B* в соседние процессы (направление пере-сылки может быть произвольным: как по возрастанию рангов процессов, так и по их убыванию).

После завершения цикла в каждом процессе будет содержаться строка *c*, равная одной из строк про-изведения *AB*. Останется переслать строки *c* главному процессу.

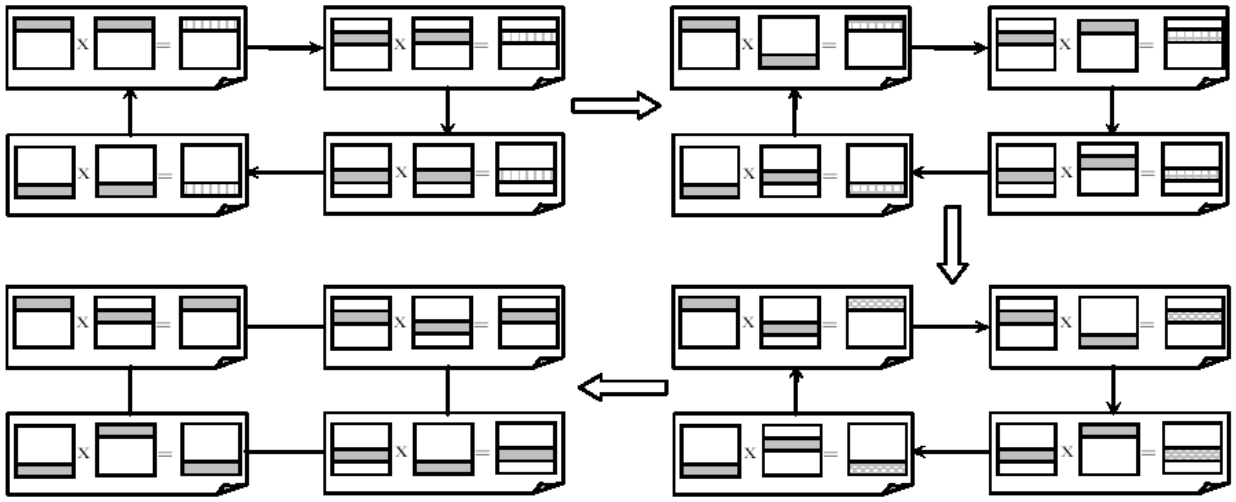
На рисунке приведена схема алгоритма 1 при условии, что циклическая пересылка столбцов матри-цы *B* выполняется в направлении убывания рангов процессов.



Ленточный алгоритм 2

Вначале производится рассылка в процесс ранга *K* элементов *K*-й строки матрицы *A* и элементов *K*-й строки матрицы *B*. Элементы строки *c*, в которой будет содержаться соответствующая строка произведе-ния *AB* результат, обнуляются.
Затем запускается цикл (число итераций равно *N*), в ходе которого выполняются два действия:
1) выполняется перемножение элементов строк матрицы *A* и матрицы *B* с одинаковыми номерами, и результаты добавляются к соответствующему элементу строки *c*;
2) выполняется циклическая пересылка строк матрицы *B* в соседние процессы (направление пересыл-ки может быть произвольным: как по возрастанию рангов процессов, так и по их убыванию).

После завершения цикла в каждом процессе будет содержаться соответствующая строка произведе-ния *AB*. Останется переслать эти строки главному процессу.
На рисунке приведена схема алгоритма 1 при условии, что циклическая пересылка строк матрицы *B* выполняется в направлении убывания рангов процессов.



Блочные алгоритмы умножения матриц

В данных алгоритмах матрицы разбиваются на блоки, представляющие собой подматрицы исходных матриц. Для простоты будем предполагать, что все матрицы являются квадратными размера *N N*, а ко-личество блоков по горизонтали и по вертикали является одинаковым и равным *q* (при этом размер всех блоков равен *K K*, где *K* = *Nq*). В этом случае операция матричного умножения может быть представле-на в блочном виде:

$$\begin{pmatrix} A_{00}A_{01}\ldots A_{0q-1} \\ \cdots \\ A_{q-10}A_{q-11}\ldots A_{q-1q-1} \end{pmatrix} \times \begin{pmatrix} B_{00}B_{01}\ldots B_{0q-1} \\ \cdots \\ B_{q-10}B_{q-11}\ldots B_{q-1q-1} \end{pmatrix} = \begin{pmatrix} C_{00}C_{01}\ldots C_{0q-1} \\ \cdots \\ c_{q-10}C_{q-11}\ldots C_{q-1q-1} \end{pmatrix}$$

При этом каждый блок C_{ij} матрицы *C* определяется как произведение соответствующих блоков мат-риц *A* и *B*:

$$C_{ij} = \sum_{s=0}^{q-1} A_{is} B_{sj}$$

При блочном разбиении данных естественно связать с каждым процессом задачу вычисления одного из блоков результирующей матрицы C. В этом случае процессу должны быть доступны все элементы со-ответствующих строк матрицы A и столбцов матрицы B. Поскольку размещение всех требуемых данных в каждом процессе приведет к их дублированию и существенному увеличению объема используемой памя-ти, необходимо организовать вычисления таким образом, чтобы в каждый момент времени процессы со-держали лишь по одному блоку матриц A и B, требуемому для расчетов, а доступ к остальным блокам обеспечивался бы при помощи передачи сообщений.
В данных алгоритмах для процессов удобно ввести двумерную декартову топологию, сопоставив каждому процессу его координаты (i, j) в этой топологии ($i, j = 0, \ldots, q$). При этом предполагается, что ко-личество процессов равно q^2.

Блочный алгоритм 1 (алгоритм Фокса)

Вначале производится рассылка в процесс с координатами (i, j) блоков A_{ij}, B_{ij} исходных матриц. Кроме того, выполняется обнуление матрицы C_{ij}, предназначенной для хранения соответствующего блока результирующего произведения AB.

Затем запускается цикл по m ($m = 0, \ldots, q$), в ходе которого выполняются три действия:

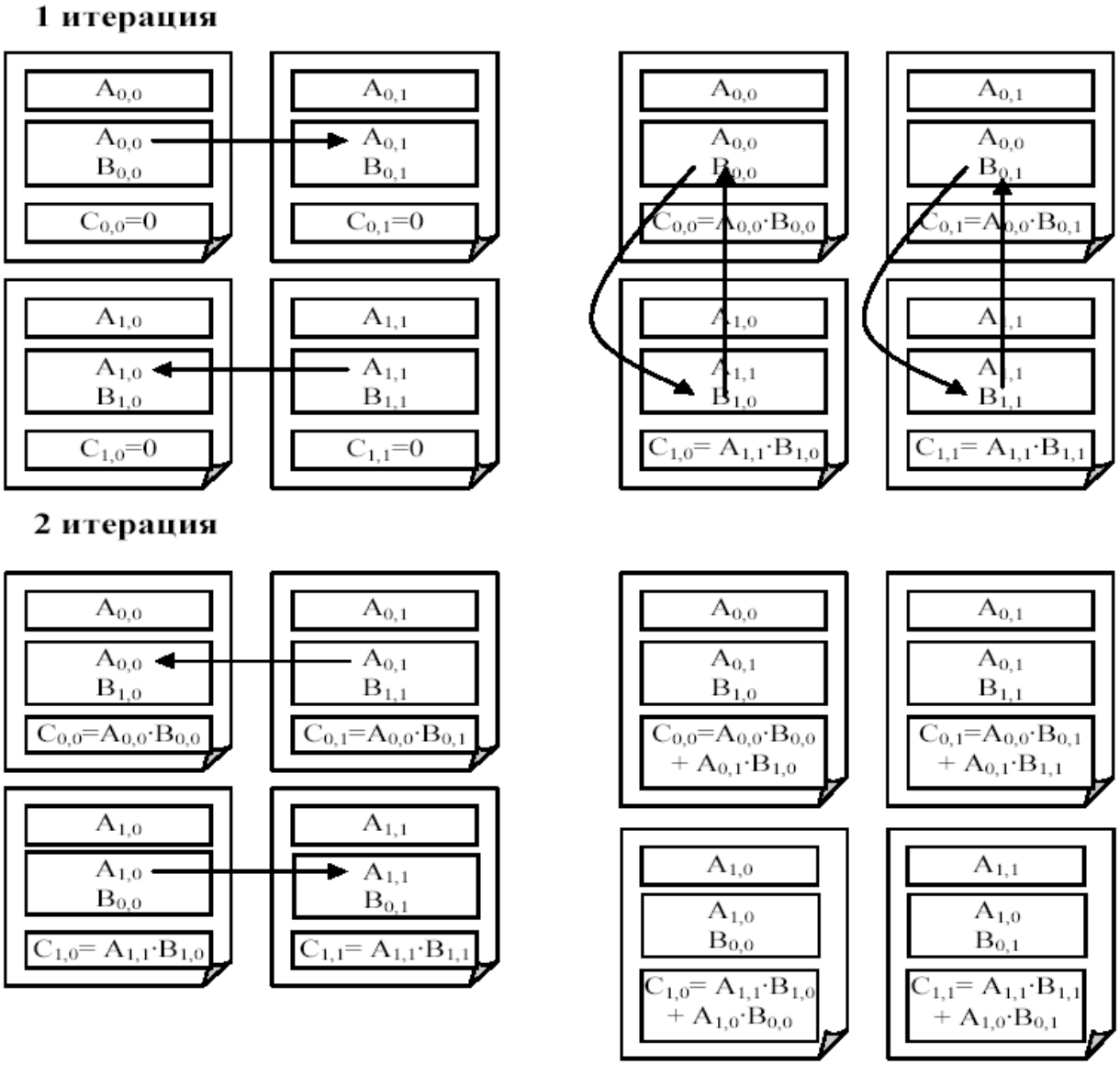
1) для каждой строки i ($i = 0, \ldots, q$) блок A_{ij} одного из процессов пересылается во все процессы этой же строки; при этом индекс j пересылаемого блока определяется по формуле $j = (i + m) \bmod q$;

2) полученный в результате подобной пересылки блок матрицы A и содержащийся в процессе (i, j) блок матрицы B перемножаются, и результат прибавляется к матрице C_{ij};

3) для каждого столбца j ($j = 0, \ldots, q$) выполняется циклическая пересылка блоков матрицы B, содер-жащихся в каждом процессе (i, j) этого столбца, в направлении убывания номеров строк.

После завершения цикла в каждом процессе будет содержаться матрица C_{ij}, равная соответствую-щему блоку произведения AB. Останется переслать эти блоки главному процессу.

На рисунке приведена схема алгоритма Фокса в случае $q = 2$.



Блочный алгоритм 2 (алгоритм Кэннона)

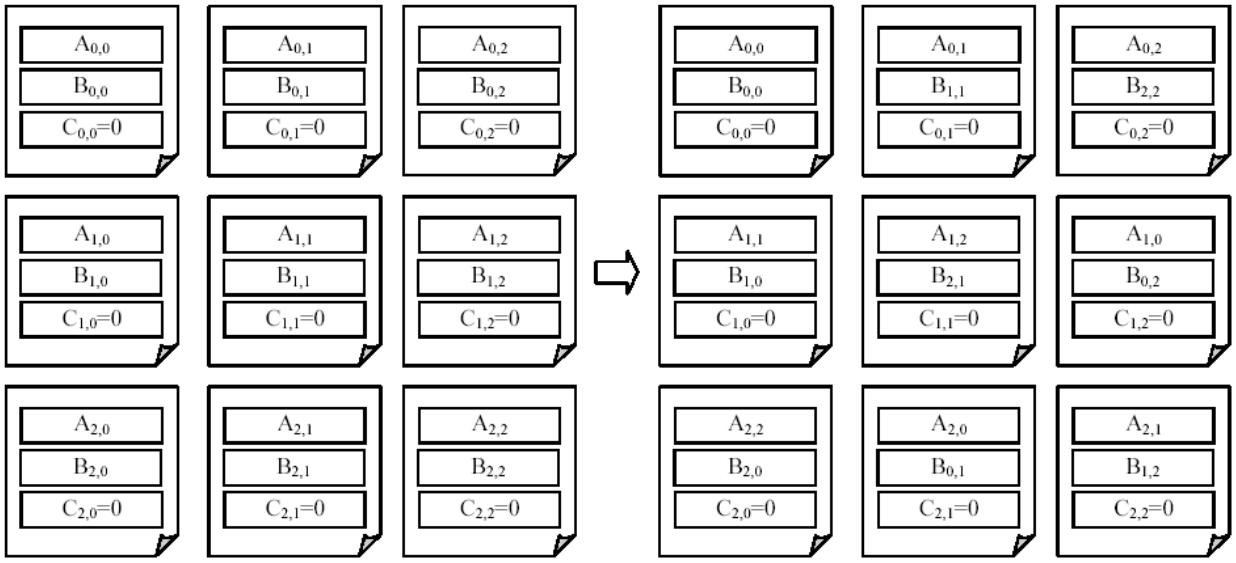
Алгоритм Кэннона отличается от алгоритма Фокса двумя аспектами. Во-первых, начальная пере-сылка блоков матриц *A* и *B* в процессы выполняется таким образом, чтобы получаемые блоки сразу могли быть перемножены без каких-либо пересылок данных. Во-вторых, при организации цикла выполняется циклическая пересылка как блоков матрицы *A* (по строкам), так и блоков матрицы *B* (по столбцам). Дей-ствия по начальной пересылке состоят из следующих шагов:

1) в каждый процесс (*i*, *j*) пересылаются блоки A_{ij} и B_{ij}, матрица C_{ij} обнуляется;

2) для каждой строки *i* декартовой решетки процессов выполняется циклический сдвиг блоков мат-рицы *A* на (*i* – 1) позиций влево (т. е. в направлении убывания номеров столбцов);

3) для каждого столбца *j* декартовой решетки процессов выполняется циклический сдвиг блоков мат-рицы *B* на (*j* – 1) позиций вверх (т. е. в направлении убывания номеров строк).

Результат подобного перераспределения блоков в случае $q = 2$ приведен на рисунке.



Затем запускается цикл из *q* итераций, в ходе которого выполняются три действия:

1) содержащиеся в процессе (*i*, *j*) блоки матриц *A* и *B* перемножаются, и результат прибавляется к матрице C_{ij};

2) для каждой строки *i* ($i = 0, \ldots, q$) выполняется циклическая пересылка блоков матрицы *A*, содержа-щихся в каждом процессе (*i*, *j*) этой строки, в направлении убывания номеров столбцов;

3) для каждого столбца *j* ($j = 0, \ldots, q$) выполняется циклическая пересылка блоков матрицы *B*, содер-жащихся в каждом процессе (*i*, *j*) этого столбца, в направлении убывания номеров строк.

После завершения цикла в каждом процессе будет содержаться матрица C_{ij}, равная соответствую-щему блоку произведения *AB*. Останется переслать эти блоки главному процессу.