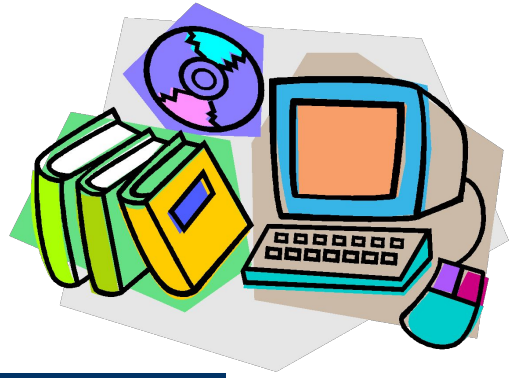
Технология DVM (1)



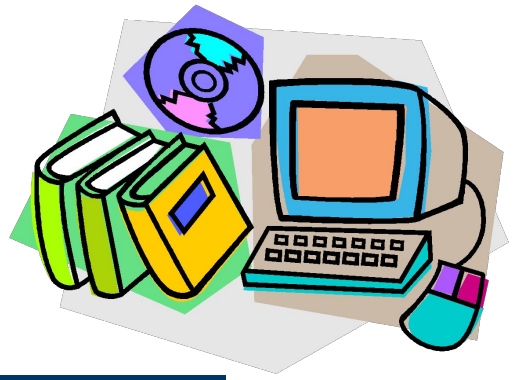
Система DVM для разработки параллельных программ создана в ИПМ РАН им. Келдыша. Модель, положенная в основу Fortran-DVM и C-DVM, объединяет элементы моделей параллелизма по данным и по управлению. DVM-система состоит из пяти основных компонентов

- компиляторы с языков Fortran-DVM и C-DVM,
- система поддержки выполнения параллельных программ,
- отладчик параллельных программ,
- анализатор производительности,
- предсказатель производительности.

При проектировании авторы опирались на следующие принципы.

- Высокоуровневая модель программирования.
- Спецификации параллелизма прозрачны для обычных компиляторов.
- Язык – расширение традиционного последовательного языка
- Динамическая настройка на параметры приложения и конфигурацию параллельного компьютера, т.е. один вариант кода для выполнения на последовательных и параллельных компьютерах разной конфигурации.

Технология DVM (2)



Параллельная программа на исходном языке DVM-фортран или DVM-C с помощью DVM-препроцессора превращается в обычную программу на фортране или на C\C++, с вызовами функций системы поддержки.

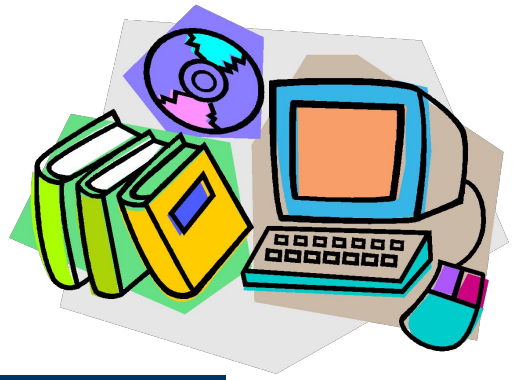
Основой для организации межпроцессорных взаимодействий в системе поддержки является MPI, поэтому программа может выполняться всюду, где есть MPI и компиляторы для C и фортрана. Это обеспечивает высокую степень переносимости программы. (В планах автоматическое преобразование DVM-программ в OpenMP-коды)

Любая DVM-программа работает в соответствии с моделью SPMD (одна последовательность команд выполняется параллельно над множеством потоков данных).

DVM-программа исполняется на виртуальной многопроцессорной системе, в качестве которой выступает группа MPI-процессов, создающихся при запуске параллельной программы на выполнение.

В момент запуска все операторы DVM-программы начинают выполняться сразу на всех процессорах виртуальной системы. В это время в DVM-программе существует единственная ветвь (единственный поток управления).

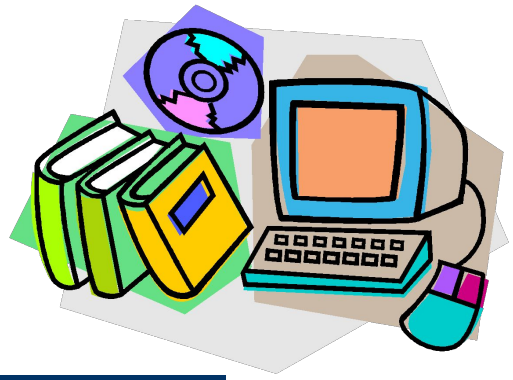
Технология DVM: организация параллелизма



Два уровня параллелизма.

- На верхнем уровне в программе выделяются с помощью директив независимые по данным крупные блоки программы, которые могут выполняться параллельно.
- На нижнем уровне, в рамках каждой ветви могут дополнительно выделяться параллельные циклы (**Cdvm$ parallel)**
- Никакой другой иерархии параллелизма DVM-программа не допускает, т.е. в теле параллельного цикла нельзя описать еще несколько независимых ветвей.
- При входе в параллельную конструкцию поток разбивается на несколько независимых потоков, каждый из которых определяет процесс вычислений на соответствующих процессорах.
- При выходе из параллельной конструкции потоки управления на всех процессорах вновь становятся одинаковыми
- Все переменные (кроме *distributed*) размножаются по всем процессорам, т.е. на каждом процессоре будет своя локальная копия каждой переменной, с которой и будет происходить работа.

DVM: реализация DSM (distributed shared memory)



- Существуют специальные «распределенные» массивы, место расположения которых определяется специальной директивой

  Cdvm$ distribute

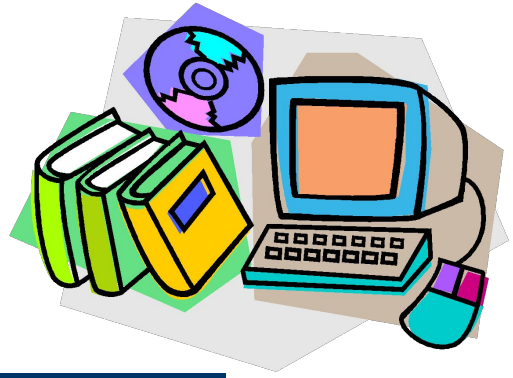
  Массив можно распределить по процессам равными порциями либо взвешенными блоками.
- Согласованное распределение массивов по процессам (напр., для работы над ними в одном цикле), осуществляется с помощью

  Cdvm$ align

  Эта директива устанавливает соответствие между компонентами массивов, которые должны быть распределены в один процессор.
- В любой момент можно динамически изменять текущее распределение данных с помощью директив

  Cdvm$ redistridute; или Cdvm$ realign.
- Организация параллелизма по данным: данные обрабатываются по правилу «левой части» (как в HPF): вычисления ведет тот процесс, где находится элемент распределенного массива с текущим индексом.

DVM: поэтапная отладка



1. Программа отлаживается в последовательном режиме
2. Программа пропускается в специальном режиме для проверки правильности и полноты DVM-директив.
3. Программа пропускается на параллельной машине в режиме сравнения промежуточных результатов ее параллельного выполнения с «эталонными» результатами последовательного выполнения.
4. Анализатор производительности позволяет получить информацию об основных показателях эффективности выполнения DVM-кода или его частей в параллельном режиме, что помогает повысить эффективность.
5. Предсказатель эффективности позволяет в последовательном режиме смоделировать выполнение DVM-кода с заданными характеристиками параллельной машины (топология связей, латентность, пропускная способность и др.) и спрогнозировать для конкретной вычислительной системы характеристики ее выполнения.

- В настоящее время система установлена и активно используется в НИВЦ МГУ, ИПМ РАН, вычислительных кластерах суперкомпьютерного центра РФ и др.

Т-система – технология автоматического динамического распараллеливания программ

Разработка была начата в Институте программных систем (Переяславль-Залесский) с конца 80х.

Характерная черта – использование парадигмы **функционального программирования** для динамического распараллеливания программ.

Функциональный стиль совмещен с традиционными языками программирования с помощью расширений языков C\C++, Фортран.

Явные параллельные конструкции в языке отсутствуют, программист нигде явно не указывает, какие части следует выполнять параллельно.

Базовые принципы Т-системы опираются на результаты общей теории функционального программирования.

Аналогия: пусть имеем сложное арифметическое выражение, состоящее из подвыражений в скобках. Эти подвыражения можно вычислять в любом порядке, в том числе параллельно. На результат это не повлияет (не считая ошибок округления).

Такой подход дает прямой метод распараллеливания функциональных программ, построенных из «**чистых функций**».

T-система:
Чистые функции

Чистая функция – одно из базовых понятий Т-системы, обозначающее функцию без побочных эффектов, т.е. функцию («подвыражение»), которая может быть выполнена параллельно с основной программой. Для обозначения чистых функций используется слово **tfun.**

В виде **tfun** в Т-системе оформляется порция работы, готовой к параллельному исполнению.

Переменные, которые должны быть возвращены как результат работы Т-функции, получают статус **неготовых переменных** (**tval**) и будут содержать **неготовые значения**.

Неготовые значения могут копироваться в другие переменные с помощью операторов присваивания. Переменные, которым присвоены неготовые значения, также получают статус tval (неготовых)

Программа выполняется, пока не встретится оператор, содержащий арифметические действия над неготовым значением. На этом операторе фрагмент программы блокируется, пока вызванная ранее чистая функция не вернет значение неготовой переменной, задействованной в операторе.

T-система:
Пример чистой функции

G(a, &x, &y, &z); вызов чистой функции,
параллельное выполнение G
результаты записываются в x,y
x,y получают статус неготовых переменных

b = x; b получает статус неготовой переменной

y = a; y получает статус готовой переменной

T = a*b выполнение блокируется до возвращения x=b

Строка «**y = a**;» вызывает сомнения: недетерминированный код? Если функция **G** вернет значение **y** уже после присвоения **y = a**; то **y** не будет равно **a**, а окажется равным значению, вычисленному функцией **G**? На самом деле в начале выполнения оператора присвоения **y = a** происходит блокировка переменной **y**. Проверяется статус **y**, и если **y** является неготовым, то выставляется отказ от ожидаемого значения. Это гарантирует **детерминированное** поведение параллельного кода.

Таким образом, **неготовые переменные являются основным средством синхронизации в T-системе**.

Т-система: неготовые переменные

- При чтении неготовой переменной происходит блокировка процесса вычисления, выполнившего такое обращение.
- При записи обычного значения в неготовую переменную она становится готовой для всех потребителей, а заблокированные ранее на данной переменной процессы выходят из состояния блокировки.

Итак, при написании параллельного кода в Т-системе программист должен описать свою программу в виде набора Т-функций на специальном диалекте ТС, что включает следующие этапы.

- разработка дизайна кода. Решается вопрос, какие именно фрагменты будут реализованы в виде Т-функций для параллельного выполнения, а какие части программы будут выполняться последовательно.
- реализация и первичная отладка на однопроцессорной системе с использованием «заглушек» для ключевых слов Т-системы.
- отладка на многопроцессорных установках.
- оптимизация.

Т-система: плюсы и минусы

+ По сравнению с другими системами параллельного программирования **преимущества** Т-системы проявляются в следующих ситуациях.
- до выполнения программы нет точной информации о том, как сбалансировать работу параллельных процессов.
- вычислительная схема близка к функциональной модели и может быть представлена как совокупность функций, вызывающих друг друга.

+ Т-система берет на себя организацию параллельных фрагментов, их распределение по узлам кластера, синхронизацию работы фрагментов, обмен данными между процессами, освобождая программиста от многих забот, традиционных для параллельного программирования.

- Вызов Т-функции может вызвать пересылку данных из одного узла кластера в другой, требующую больших накладных расходов.
- Балансировка нагрузки на узлы возложена на Т-систему, но не гарантирована оптимальность: дробление на слишком маленькие порции, реализуемые в Т-функциях, приводит к росту накладных расходов; в то время как слишком большая вычислительная сложность, может привести к малому числу порождаемых параллельных фрагментов, и, как следствие, к низкой эффективности.

Язык НОРМА – непроцедурное описание разностных моделей алгоритмов

НОРМА – специализированный непроцедурный язык, предназначенный для спецификации задач вычислительного характера (в частности, для задач математической физики).

Все конструкции языка носят декларативный характер и описывают правила вычисления значения. Основное назначение языка – автоматизация процесса разработки программ. Программист работает «почти» в терминах математических формул, что значительно упрощает его работу.

При этом, естественно, задача транслятора усложняется, поскольку помимо традиционных задач синтаксического и семантического анализа он выполняет сам синтез программы.

Идеи автоматического построения программы по спецификации задачи сформулированы еще в 1963. Их развитие привело к появлению языка НОРМА и нескольких трансляторов для разных платформ

Разработчик прикладных программ абстрагируется от особенностей конкретных компьютеров и мыслит в категориях своей предметной области.

Язык НОРМА

Отталкиваясь от потребностей ИПМ им. Келдыша разработчики старались максимально упростить программирование численного решения определенных классов задач математической физики.

Специфика предметной области – ориентация на сеточные методы, что наложило отпечаток на концепцию и все конструкции языка.

В программе на языке НОРМА не требуется информации о порядке выполнения операций. Порядок предложений языка может быть произвольным. Язык позволяет формулировать запрос на вычисления, не уточняя, каким именно образом эти вычисления организовывать.

Все информационные связи учитываются транслятором-синтезатором на этапах анализа исходной программы и синтеза выходного текста. На транслятор ложится и выбор конкретного способа организации вычислений – в частности, на этапе синтеза результирующей программы транслятор может сгенерировать как последовательный, так и параллельный код. Выходные программы генерируются для последовательных и параллельных вычислительных систем с общей и распределенной памятью на fortran MPI, fortran PVM, Fortran 77 и др.

Язык НОРМА: разделы и области

Программа на языке НОРМА состоит из одного или нескольких **разделов**.

- Разделы могут быть трех видов – главный раздел, простой раздел и раздел-функция.
- Разделы могут вызывать друг друга по имени и передавать данные при помощи описаний INPUT и OUTPUT. Рекурсивные вызовы разделов запрещены.
- Все имена, описанные в разделе, локализованы в нем. Понятия глобальных переменных нет. Список формальных параметров делится ключевым словом RESULT на входные (слева от RESULT) и на выходные (справа) параметры. В разделе-функции слово RESULT не используется, все параметры – входные.
- В теле раздела задаются описания, операторы, итерации. Порядок их расположения – произвольный.
- Базовое понятие языка НОРМА – **область**. Это совокупность целочисленных наборов, задающих координаты в n-мерном индексном пространстве. Области могут быть одно- и многомерными.

Язык НОРМА: операторы

Задав область, можно определить ряд **величин на области**. Тем самым, запись сеточных выражений упрощается и близка к формульной (не нужны циклы)

В языке НОРМА определены три вида операторов – **скалярный оператор, оператор ASSUME, вызов раздела.**

- Скалярный оператор предназначен для вычисления арифметических значений скаляров. Аналог оператора присваивания в традиционных языках программирования, где слева указывается имя скалярной переменной, а справа – скалярное арифметическое выражение.
- Оператор ASSUME используется для вычисления арифметических значений величин на областях. Однозначно определяет правило вычисления значения величины, но не требует немедленного выполнения вычислений в данном месте программы, не задает порядка и способа вычислений. Оператор не содержит никакой информации о последовательном или параллельном исполнении.
- Режим вычислений определяется последовательностью вызовов разделов программы.

Язык НОРМА: Resumé

В рамках языка Норма предусмотрена также специальная **конструкция ITERATION** для организации итерационного процесса.

Итак, программа на языке Норма представляет собой формализованное, описание вычислительной схемы для решения математической задачи с использованием сеточных и итерационных методов. Это описание существенно проще по сравнению с традиционными языками, где для сеточных вычислений и итерационных процессов требуется организация большого количества циклов, введения соответствующих индексов, использования условных операторов и др.

+ Направление перспективно с точки зрения автоматизации распараллеливания, т.к. в основу НОРМА положены математические формулы, ⇒ выявить и реализовать параллелизм существенно проще, чем в случае с уже готовым последовательным кодом.

- Скрыта от программиста реальная последовательность операторов ⇒ усложняется отладка.

- Программист полностью зависит от транслятора и не может повлиять на эффективность создаваемого параллельного кода.

Процедуры MPI повторение и дополнение

«ПРОЖИТОЧНЫЙ МИНИМУМ»

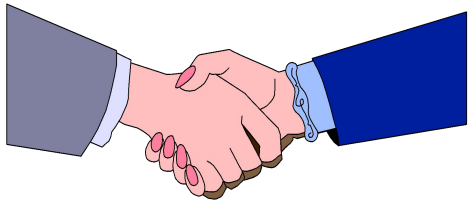
MPI_Init **инициализация параллельной части программы**
MPI_Finalize **завершение параллельной части программы**
MPI_Comm_size **число процессов в группе**
MPI_Comm_rank **номер процесса**
MPI_Send **блокирующая передача данных**
MPI_Recv **блокирующий прием данных**

Процедуры общего назначения

MPI_Wtime: возвращает астрономическое время в сек.
MPI_Abort: завершение работы всей группы
MPI_Get_processor_name: имя узла, где запущен вызвавший процесс

Обмен «point-to-point:

MPI_Sendrecv: Операция объединяет в едином запросе посылку и прием сообщений



Операции “Point-to-Point”: асинхронный обмен

MPI_Isend: Передача сообщения, аналогичная *MPI_Send*, однако возврат происходит сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в *buf*. Это означает, что нельзя использовать данный буфер для других целей без получения информации о завершении данной посылки.

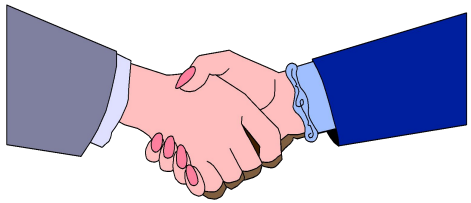
MPI_Irecv: Прием сообщения, аналогичный *MPI_Recv*, однако возврат происходит сразу после инициализации процесса приема без ожидания приема всего сообщения в *buf*.

Окончание процессов приема\передачи, когда можно использовать *buf,* можно определить с помощью специального параметра *request* и процедур ***MPI_Wait* и *MPI_Test***.

Сообщение, отправленное любой из процедур *MPI_Send* и *MPI_Isend*, может быть принято любой из процедур *MPI_Recv* и *MPI_Irecv*.

MPI_Wait: Ожидание завершения асинхронных процедур *MPI_Isend* или *MPI_Irecv*, ассоциированных с идентификатором *request*.

MPI_Test: Проверка завершенности процедур *MPI_Isend* или *MPI_Irecv*. Параметр *flag*=1, если операция завершена, и 0 в противном случае.



Операции MPI “Point-to-Point”:
Объединение запросов на взаимодействие

Несколько запросов на прием и/или передачу могут объединяться вместе для того, чтобы далее их можно было бы запустить одной командой для снижения накладных расходов на обработку обмена.

Способ приема сообщения никак не зависит от способа его посылки: сообщение, отправленное с помощью объединения запросов либо обычным способом, может быть принято как обычным способом, так и с помощью объединения запросов.

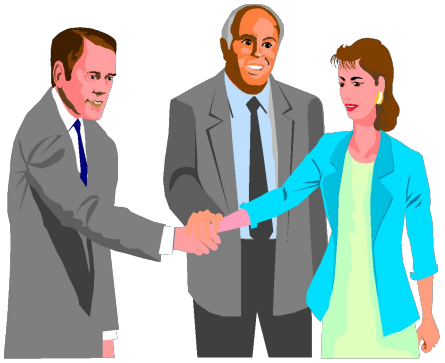
MPI_Send_init: Формирование запроса на выполнение пересылки данных. Все параметры такие же, как у *MPI_Isend*, однако пересылка не начинается до вызова *MPI_Startall*.

MPI_Recv_init: Формирование запроса на выполнение приема данных. Все параметры такие же, как у *MPI_Irecv*, однако реальный прием не начинается до вызова подпрограммы *MPI_Startall*.

MPI_Startall: Запуск всех отложенных взаимодействий, ассоциированных вызовами *MPI_Send_init* и *MPI_Recv_init*.

Все взаимодействия запускаются в режиме без блокировки, а их завершение можно с помощью процедур *MPI_Wait* и *MPI_Test*.

Коллективные операции MPI

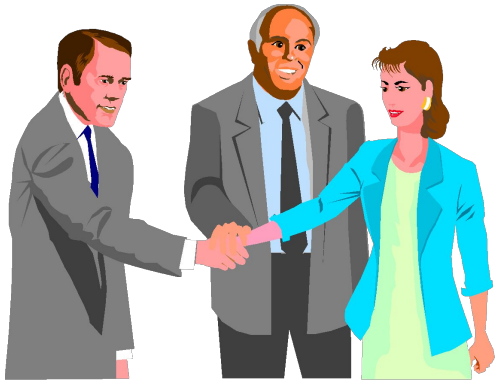


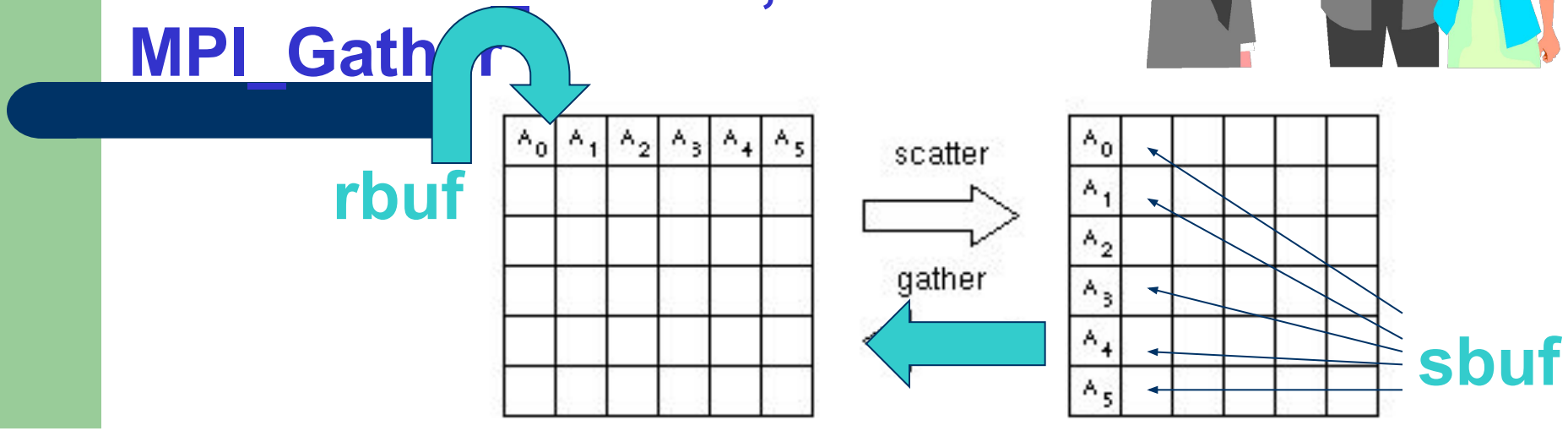
В операциях коллективного взаимодействия процессов участвуют все процессы коммуникатора. Соответствующая процедура должна быть вызвана каждым процессом, быть может, со своим набором параметров.

Возврат из процедуры коллективного взаимодействия может произойти, когда участие процесса в данной операции уже закончено. Возврат означает, что разрешен свободный доступ к буферу приема\посылки, но не означает, что операция завершена другими процессами.

- **MPI_Barrier(comm) – барьерная синхронизация**
- **MPI_Bcast – рассылка данных всем процессам**
- **MPI_Gather – сборка данных**
- **MPI_Scatter – рассылка сегментов массива всем процессам**
- **MPI_Reduce, MPI_AllReduce – глобальные операции**

Коллективные операции
MPI_Scatter;
MPI_Gather





int **MPI_Gather** (void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, int dest, MPI_Comm comm)

sbuf - адрес начала буфера посылки

scount и *stype* - число и тип элементов в посылаемом сообщении

OUT *rbuf* - адрес начала буфера сборки данных

Rcount u rtype - число и тип элементов в принимаемом сообщении

dest - номер процесса, на котором происходит сборка данных

Сборка данных со всех процессов в буфер *rbuf* процесса *dest*. Каждый процесс, включая *dest*, посылает содержимое своего буфера *sbuf* процессу *dest*. Данные в *rbuf* располагаются в порядке возрастания номеров процессов. Параметр *rbuf* имеет значение только на *dest.*

Группы и коммуникаторы

В MPI существуют широкие возможности для операций над группами процессов и коммуникаторами. Это бывает необходимо в случаях:

Во-первых, чтобы дать возможность некоторой группе процессов работать над своей независимой подзадачей.

Во-вторых, если особенность алгоритма такова, что только часть процессов должна обмениваться данными, бывает удобно завести для их взаимодействия отдельный коммуникатор.

В-третьих, при создании библиотек подпрограмм нужно гарантировать, что пересылки данных в библиотечных модулях не пересекутся с пересылками в основной программе. Решение этих задач можно обеспечить в полном объеме только при помощи создания нового независимого коммуникатора.

Группа – упорядоченное множество процессов. Каждому процессу в группе сопоставлено целое число (***ранг***). Базовая группа связана с коммуникатором **MPI_COMM_WORLD**, в нее входят все процессы приложения.

Операции с группами процессов

Новые группы можно создавать как на основе уже существующих групп, так и на основе коммуникаторов

MPI_Comm_group(COMM, GROUP). Получение группы GROUP, соответствующей коммуникатору COMM.

MPI_Group_incl(GROUP, N, RANKS, NEWGROUP). Создание группы NEWGROUP из N процессов прежней группы GROUP с рангами RANKS

MPI_Group_intersection: Создание группы из пересечения двух групп.

MPI_Group_union: Создание группы путем объединения двух групп.

MPI_Group_difference: Создание новой группы как разности двух групп.

MPI_Group_size(GROUP, SIZE). Определение количества SIZE процессов в группе GROUP.

MPI_Group_rank(GROUP, RANK). Определение номера процесса RANK в группе GROUP

MPI_Group_free(GROUP).

Операции с коммуникаторами

Коммуникатор предоставляет возможность независимых обменов данными в отдельной группе. Каждой группе процессов может соответствовать несколько коммуникаторов, но каждый в любой момент времени однозначно соответствует только одной группе. Следующие коммуникаторы создаются сразу после вызова процедуры **MPI_Init**:

MPI_COMM_WORLD – коммуникатор, объединяющий все процессы;

MPI_COMM_NULL – ошибочный коммуникатор;

MPI_COMM_SELF – коммуникатор включает только вызвавший процесс.

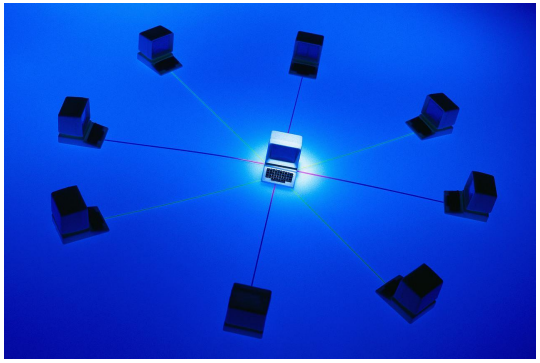
MPI_Comm_dup(COMM, NEWCOMM). ***Создание*** нового коммуникатора NEWCOMM с той же группой процессов и атрибутами, что и у коммуникатора COMM.

MPI_Comm_create. ***Создание*** нового коммуникатора из имеющегося коммуникатора для группы процессов, которая является подмножеством группы, связанной с существующим коммуникатором.

MPI_Comm_split. ***Разбиение*** коммуникатора на несколько новых.

MPI_Comm_free(COMM). ***Удаление*** коммуникатора COMM.

Виртуальные топологии



Топология – механизм сопоставления процессам некоторого коммуникатора альтернативной схемы адресации. Топология используется программистом для более удобного обозначения процессов, и таким образом, приближения параллельной программы к структуре математического алгоритма. Топология может использоваться системой для оптимизации распределения процессов по физическим процессорам используемого параллельного компьютера при помощи изменения порядка нумерации процессов внутри коммуникатора.

Декартова топология (прямоугольная решетка произвольной размерности)

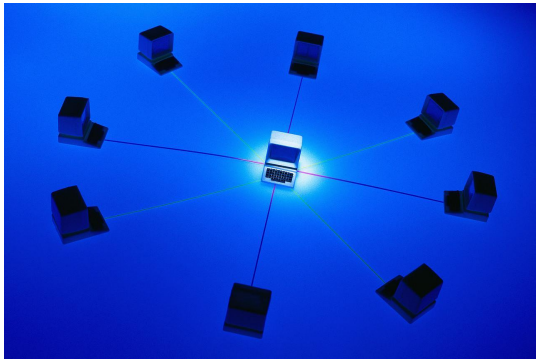
Топология графа.

MPI_Topo_test. Процедура определения типа топологии.

Декартова топология

MPI_Cart_create: Создание коммуникатора, обладающего декартовой топологией, из процессов существующего коммуникатора с заданной размерностью получаемой декартовой решетки

Виртуальные топологии



Каждому процессу ставится в соответствие набор индексов - декартовых координат в соответствии с размерностью задаваемой топологии. Если топология трехмерная – каждому процессу соответствует набор (I,j,k), определяющий его место в виртуальной решетке.

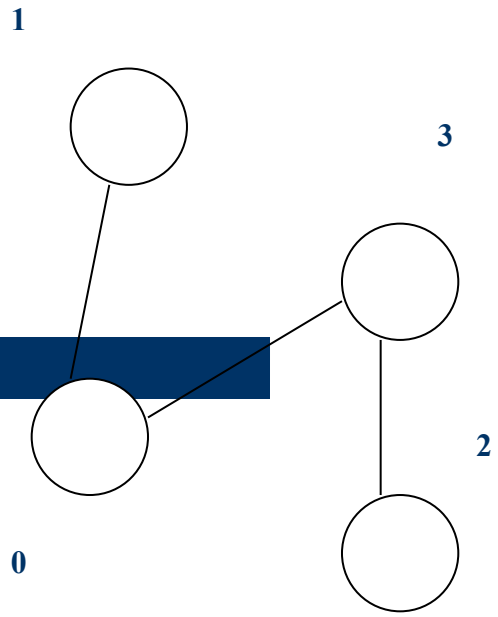
Некоторые возможности операций с декартовой топологией.

- Определение декартовых координат процесса по его рангу в коммуникаторе.
- Определение ранга процесса в коммуникаторе по его декартовым координатам.
- Расщепление коммуникатора, с которым связана декартова топология, на подгруппы (декартовым подрешеткам меньшей размерности).

Топология графа

MPI_Graph_create. Создание на основе существующего коммуникатора нового коммуникатора с топологией графа с заданным числом вершин. INDEX содержит суммарное количество соседей для первых I вершин. EDGES содержит упорядоченный список номеров процессов-соседей всех вершин.

Виртуальные топологии



Граф определяется количеством вершин и списком их соседей

Процесс	Соседи
0	1, 3
1	0
2	3
3	0, 2

INDEX=2, 3, 4, 6;
EDGES=1, 3, 0, 3, 0, 2

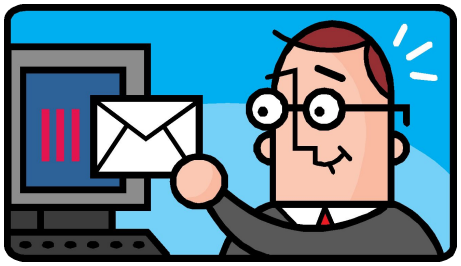
INDEX содержит суммарное количество соседей для первых I вершин.

EDGES содержит упорядоченный список номеров процессов-соседей всех вершин.

Некоторые возможности операций с графовой топологией.

- Определение количества непосредственных соседей процесса с заданным рангом.
- Определение рангов непосредственных соседей процесса с заданным рангом.
- Определение числа вершин и числа ребер графовой топологии, связанной с данным коммуникатором.

Пересылка разнотипных данных



Под *сообщением* в MPI понимается массив *однотипных* данных, расположенных в последовательных ячейках памяти.

Часто в программах требуются пересылки более сложных объектов данных, состоящих из *разнотипных* элементов или расположенных не в последовательных ячейках памяти.

В этом случае можно либо посылать данные небольшими порциями расположенных подряд элементов одного типа, либо использовать копирование данных перед отсылкой в некоторый промежуточный буфер.

НО: оба варианта являются достаточно неудобными и требуют дополнительных затрат как времени, так и оперативной памяти.

Поэтому пересылки разнотипных данных в MPI предусмотрены два специальных способа:

- *Производные типы данных*;
- *Упаковка данных*.

Пересылка разнотипных данных: Производные типы данных

Производные типы данных создаются во время выполнения программы с помощью процедур-конструкторов на основе существующих к моменту вызова конструктора типов данных.

Создание типа данных состоит из двух этапов:

- Конструирование типа.
- Регистрация типа.

После регистрации производный тип данных можно использовать наряду с предопределенными типами в операциях пересылки, в том числе и в коллективных операциях. После завершения работы с производным типом данных его рекомендуется аннулировать. При этом все произведенные на его основе новые типы данных остаются и могут использоваться дальше.

Производный тип данных характеризуется последовательностью базовых типов данных и набором целочисленных значений смещения элементов типа относительно начала буфера обмена. Смещения могут быть как положительными, так и отрицательными, не обязаны различаться, не требуется их упорядоченность.

Пересылка разнотипных данных: Производные типы данных

Таким образом, последовательность элементов данных в производном типе может отличаться от последовательности исходного типа, а один элемент данных может встречаться в конструируемом типе многократно.

MPI_Type_contiguous(COUNT, TYPE, NEWTYPE). ***Создание*** нового типа данных NEWTYPE, состоящего из COUNT последовательно расположенных элементов базового типа TYPE. Фактически новый тип данных представляет массив данных базового типа как отдельный объект.

MPI_Type_vector(COUNT, BLOCKLEN, STRIDE, TYPE, NEWTYPE). ***Создание*** типа NEWTYPE, состоящего из COUNT блоков по BLOCKLEN элементов базового типа TYPE. Следующий блок начинается через STRIDE элементов базового типа после начала предыдущего блока.

MPI_Type_struct – создание структурного типа данных.

MPI_Type_commit – регистрация созданного производного типа данных.

MPI_Type_size – определение размера типа данных в байтах.

MPI_Type_free – аннулирование производного типа данных.

Пересылка разнотипных данных: Упаковка данных

Для пересылок разнородных данных типов можно использовать **операции упаковки и распаковки данных.** Разнородные или расположенные не в последовательных ячейках памяти данные помещаются в один непрерывный буфер, который и пересылается, далее полученное сообщение снова распределяется по нужным ячейкам памяти.

MPI_Pack(INBUF,INCOUNT,TYPE,OUTBUF,OUTSIZE,POSITION,COMM).
Упаковка INCOUNT элементов типа TYPE из массива INBUF в массив OUTBUF со сдвигом POSITION байт от начала массива. После выполнения процедуры параметр POSITION увеличивается на число байт, равное размеру записи. Параметр COMM указывает на коммуникатор, в котором в дальнейшем будет пересылаться сообщение. Для пересылки упакованных данных используется тип данных **MPI_PACKED**.

MPI_Unpack(INBUF,INSIZE,POSITION,OUTBUF,OUTCOUNT,TYPE,COMM).
Распаковка OUTCOUNT элементов TYPE из массива INBUF со сдвигом POSITION байт от начала массива в массив OUTBUF. Массив INBUF имеет размер не менее INSIZE байт.

MPI_Pack_size – определение необходимого для упаковки объема памяти