МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П.КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)»

С. Б. Попов

Инструментальные средства и технологии параллельного программирования

Презентации к курсу лекций

Самара

2011

Автор: ПОПОВ Сергей Борисович

Курс лекций посвящен разработке и отладке параллельных приложений для многопроцессорных вычислительных систем различной архитектуры. Курс «Инструментальные средства и технологии параллельного программирования» предназначен для магистров факультета информатики, обучающихся по направлению "Прикладная математика и информатика" в рамках магистерской программы 010400.68 – "Технологии параллельного программирования и суперкомпьютинг".

Параллельное программирование

«Прикладная математика и информатика»
направление 010400.68

магистерская программа

Технологии параллельного программирования и суперкомпьютинг

Структура курса

- Лекции: 36 часов
- Лабораторные работы: 36 часа
- Самостоятельная работа: 72 + 36 час.

- ИТОГ: Экзамен

Темы

- основные архитектуры параллельных вычислительных систем, их классификация
- основные модели параллельного программирования
- основы методологии проектирования параллельных программ
- методы оценки производительности параллельных программ
- основы программирования с использованием библиотеки MPI
- основы программирования на OpenMP

Литература

- В.В.Воеводин, Вл.В.Воеводин
  Параллельные вычисления
  СПб: БХВ-Петербург, 2002

- Богачев К.Ю.
  Основы параллельного программирования
  М.: БИНОМ. Лаборатория знаний, 2003

Литература (библиотека)

- **Введение в программирование для параллельных ЭВМ и кластеров**
  Учебн. Пособие
  Авт.-сост.: *Кравчук В.В., Попов С.Б., Привалов А.Ю., Фурсов В.А., Шустов В.А.*
  Под ред. *В.А. Фурсова*.
  Самара: СНЦ РАН, СГАУ, 2000. 87 с.
- **Начальные сведения для работы на вычислительном кластере**
  Метод. указания для курс. и дипл. проектир. и учеб.-исслед. работы студентов
  Авт.-сост.: С.Б. Попов, С.А. Скуратов, В.А. Фурсов.
  – Самара: СГАУ, 2004. 20 с.

Литература (internet)

- В.Н. Дацюк, А.А. Букатов, А.И. Жегуло **Методическое пособие по курсу "Многопроцессорные системы и параллельное программирование"**
  http://rsusu1.rnd.runnet.ru/koi8/index.html
- **MPI: The Message Passing Interface**
  http://parallel.ru/tech/tech_dev/mpi.html
- **http://www.parallel.ru**

Задачи «Большого вызова»

- Моделирование вселенной
- Моделирование климата
- Моделирование белков и др. химических объектов
- Моделирование мозга (нейронные сети)
- Расчеты сложных физических реакций

Параллелизм

- Стратегическое направление развития вычислительной техники
- Основной путь повышения производительности компьютеров
- ASCI – Accelerated Strategic Company Initiative

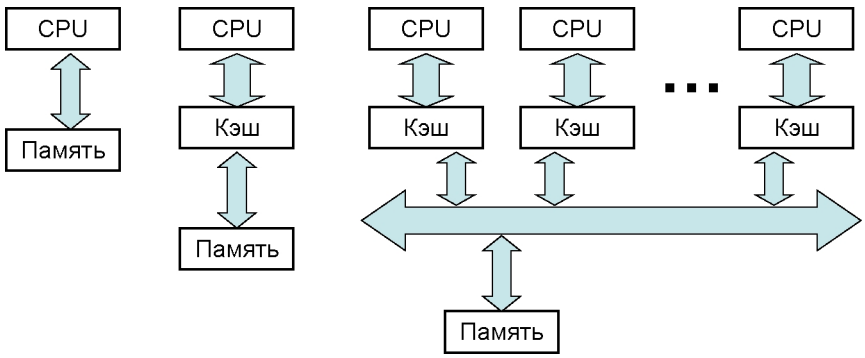
Пути достижения параллелизма

- Независимость функционирования отдельных устройств компьютеров
- Избыточность элементов вычислительной системы (использование специализированных устройств и дублирование)
- Конвейерная реализация обрабатывающих устройств

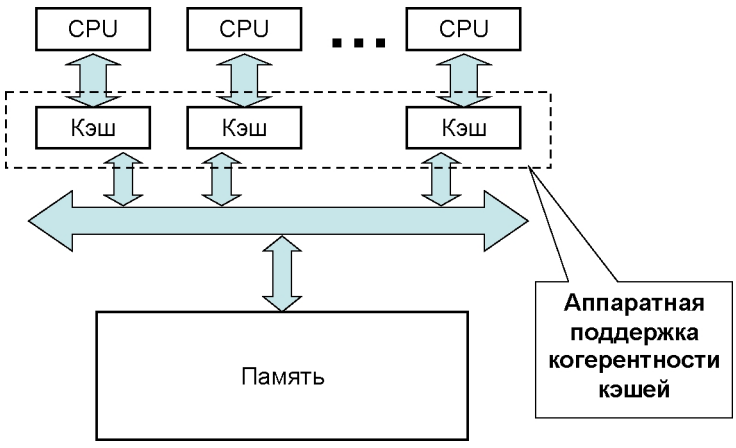
Сдерживающие факторы

- Высокая стоимость параллельных вычислительных систем (суперкомпьютеров)
- Необходимость обобщения последовательных программ
- Потери производительности при организации параллелизма (гипотеза Минского)
- Постоянное совершенствование последовательных компьютеров
- Последовательность большинства вычислительных алгоритмов (закон Амдала)
- Зависимость эффективности параллелизма от учета архитектуры параллельных систем

Вперед
к параллельному компьютеру



Симметричные мультипроцессорные системы SMP-системы



SMP-системы: архитектура

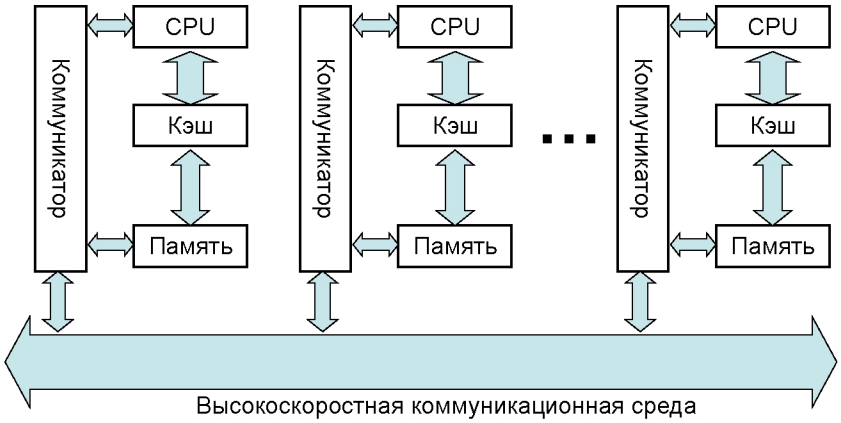
Система состоит из нескольких однородных процессоров и массива общей памяти (обычно из нескольких независимых блоков). Все процессоры имеют доступ к любой точке памяти с одинаковой скоростью. Процессоры подключены к памяти либо с помощью общей шины (базовые 2-4 процессорные SMP-сервера), либо с помощью crossbar-коммутатора (HP 9000). Аппаратно поддерживается когерентность кэшей.

SMP-системы: Масштабируемость

- Наличие общей памяти сильно упрощает взаимодействие процессоров между собой, однако накладывает сильные ограничения на их число - не более 32 в реальных системах. Для построения масштабируемых систем на базе SMP используются кластерные или NUMA-архитектуры.

- **Операционная система**
  Вся система работает под управлением единой ОС (обычно UNIX-подобной). ОС автоматически (в процессе работы) распределяет процессы/нити по процессорам (scheduling), но иногда возможна и явная привязка.
- **Модель программирования**
  Программирование в модели **общей памяти**. (POSIX threads, OpenMP). Для SMP-систем существуют сравнительно эффективные средства автоматического распараллеливания.

Укрупненно-распараллеленные системы (MPP-системы)



MPP-системы: архитектура

Система состоит из однородных ***вычислительных узлов***, включающих:

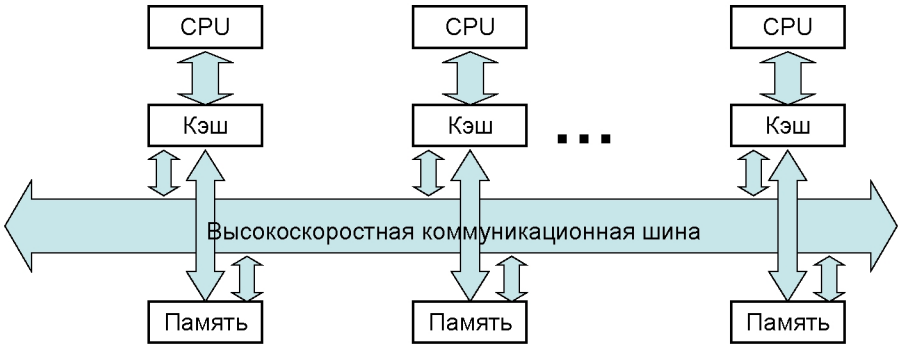
- один или несколько центральных процессоров
- локальную память (прямой доступ к памяти других узлов невозможен)
- коммуникационный процессор или сетевой адаптер
- иногда - жесткие диски (как в SP) и/или другие устройства В/В

К системе могут быть добавлены специальные узлы ввода-вывода и управляющие узлы. Узлы связаны через некоторую коммуникационную среду (высокоскоростная сеть, коммутатор и т.п.)

- **Масштабируемость**
  Общее число процессоров в реальных системах достигает нескольких тысяч (ASCI Red, Blue Mountain).
- **Операционная система** Существуют два основных варианта:
  - Полноценная ОС работает только на управляющей машине (front-end), на каждом узле работает сильно урезанный вариант ОС, обеспечивающие только работу расположенной в нем ветви параллельного приложения. Пример: Cray T3E.
  - На каждом узле работает полноценная UNIX-подобная ОС (вариант, близкий к кластерному подходу). Пример: IBM RS/6000 SP + ОС AIX, устанавливаемая отдельно на каждом узле.
- **Модель программирования**
  Программирование в рамках модели **передачи сообщений** (MPI, PVM, BSPlib)

Системы с неоднородным доступом к памяти (NUMA)

- cc-NUMA – cashe-coherent NUMA



NUMA-системы: архитектура

Система состоит из однородных базовых модулей (плат), состоящих из небольшого числа процессоров и блока памяти. Модули объединены с помощью высокоскоростного коммутатора.
Поддерживается единое адресное пространство, аппаратно поддерживается доступ к удаленной памяти, т.е. к памяти других модулей. При этом доступ к локальной памяти в несколько раз быстрее, чем к удаленной.

NUMA-системы: масштабируемость

Масштабируемость NUMA-систем ограничивается

- объемом адресного пространства
- возможностями аппаратуры поддержки когерентности КЭШей
- возможностями операционной системы по управлению большим числом процессоров

На настоящий момент, максимальное число процессоров в NUMA-системах составляет 256

- **Операционная система**
  Обычно вся система работает под управлением единой ОС, как в SMP. Но возможны также варианты динамического "подразделения" системы, когда отдельные "разделы" системы работают под управлением разных ОС (например, Windows и UNIX в NUMA-Q 2000).

- **Модель программирования**
  Аналогично SMP.

Параллельные векторные системы PVP-системы

- Основным признаком PVP-систем является наличие специальных векторно-конвейерных процессоров, в которых предусмотрены команды однотипной обработки векторов независимых данных, эффективно выполняющиеся на конвейерных функциональных устройствах.
- Как правило, несколько таких процессоров (1-16) работают одновременно над общей памятью (аналогично SMP) в рамках многопроцессорных конфигураций. Несколько таких узлов могут быть объединены с помощью коммутатора (аналогично MPP).

PVP-системы

- **Модель программирования**
  Эффективное программирование подразумевает ***векторизацию циклов*** (для достижения разумной производительности одного процессора)
  и их ***распараллеливание*** (для одновременной загрузки нескольких процессоров одним приложением).

Кластерные системы

- Набор рабочих станций (или даже ПК) общего назначения, используется в качестве дешевого варианта укрупненно-распараллеленного компьютера.
- Для связи узлов используется одна из стандартных сетевых технологий (Fast/ Gigabit Ethernet, Myrinet) на базе шинной архитектуры или коммутатора.
- **Гетерогенные** (неоднородные) кластеры

- **Операционная система**
  Используются стандартные для рабочих станций ОС, чаще всего, свободно распространяемые - Linux/FreeBSD, вместе со специальными средствами поддержки параллельного программирования и распределения нагрузки.
- **Модель программирования**
  Программирование, как правило, в рамках модели передачи сообщений (чаще всего - MPI).

- Дешевизна подобных систем оборачивается большими накладными расходами на взаимодействие параллельных процессов между собой, что сильно сужает потенциальный класс решаемых задач.

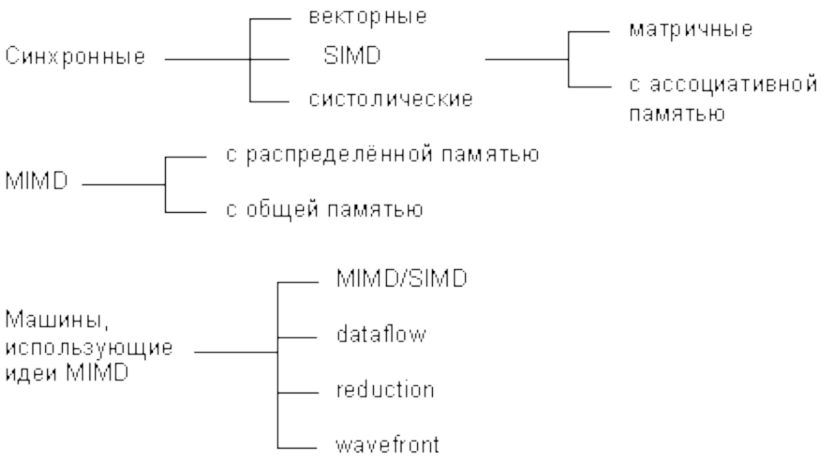
Классификация вычислительных систем
Систематика Флинна (Flynn)

- SISD – Single Instruction, Single Data
обычные последовательные компьютеры
- SIMD - Single Instruction, Multiple Data
большинство векторных компьютеров
- MIMD - Multiple Instruction, Multiple Data
параллельные многопроцессорные вычислительные системы
- MISD – не существует

Классификация Дункана

- ***Параллельная архитектура*** - это такой способ организации вычислительной системы, при котором допускается, чтобы множество процессоров (простых или сложных) могло бы работать одновременно, взаимодействуя по мере надобности друг с другом

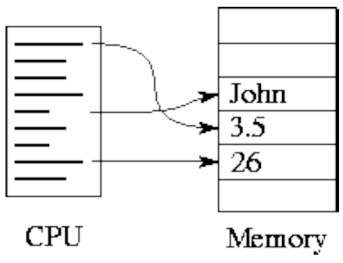
Классификация Дункана



Модели параллельного программирования

Лекция 2

Модель компьютера



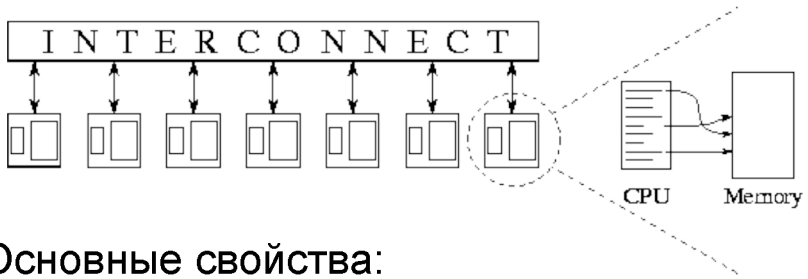
read & write

Свойства модели:

- Простота
- Реалистичность

Последовательный характер выполнения программ

Модель параллельного компьютера



Основные свойства:

- Параллельность
- Масштабируемость
- Локальность (read/write vs send/recv)

Модель последовательного программирования

Основное свойство:

- Модульность
  - Процедуры
  - Структуры данных
  - ОО-конструкции
  - т.п.

Основные свойства модели параллельного программирования

- Параллелизм (concurrency)
  способность выполнять несколько действий (операций, задач) одновременно
- Масштабируемость (scalability)
  потенциальные возможности увеличения производительности при увеличении числа процессоров
- Локальность (locality)
  предпочтительность использования локальной памяти по сравнению с обращением к удаленной памяти
- Модульность (modularity)
  возможность декомпозиции сложных программ на простые компоненты

Критерии выбора модели

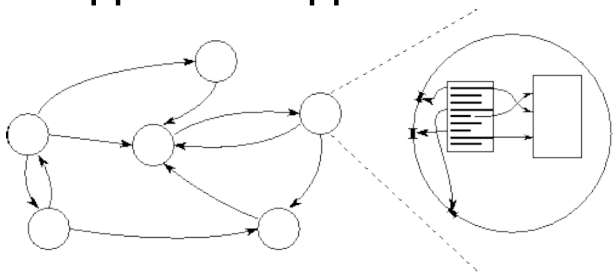
- Удобные механизмы абстракции для обсуждения параллелизма и локализации, которые способствовали бы разработке масштабируемых и модульных программ.
- Абстракции должны быть достаточно простыми в применении и соответствовать архитектуре параллельных компьютеров.

Модели параллельного программирования

- Модель «задача/канал»
- Модель передачи сообщений
- Модель разделяемой общей памяти
- Модели, использующие параллелизм данных

Параллелизм задач / Параллелизм данных

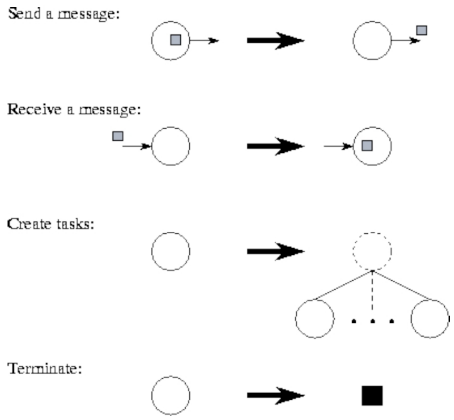
Модель «задача/канал»



1. Параллельная программа состоит из одной или нескольких задач. Задачи выполняются одновременно. Число задач может изменяться при выполнении программы.
2. Задача инкапсулирует последовательную программу и локальную память. Кроме того, набор входных (inports) и выходных (outports) портов определяет интерфейс задачи с ее окружением.

Модель «задача/канал»

3. Задача может выполнять четыре основных действия в дополнение к чтению/записи локальной памяти:
   - послать сообщения на outports
   - получить сообщения из inports
   - создать новые задачи
   - завершить выполнение



Модель «задача/канал»

4. Операция посылки сообщения является асинхронной: она завершается немедленно. Операция получения сообщения является синхронной: ее выполнение блокирует задачу до прихода сообщения.
5. Пары входных/выходных портов могут быть связаны очередью сообщений - *программным каналом*. Каналы могут создаваться и удаляться, ссылки на каналы могут передаваться в сообщениях, что делает соединения динамическими.
6. Задачи могут быть отображены на имеющиеся процессоры различными способами; используемое отображение не влияет на семантику программы. В частности несколько задач могут быть отображены на единственный процессор.

Модель «задача/канал»

- Абстракция *задача* обеспечивает механизм локальности (локальные данные и удаленные – есть различия).
- Абстракция *канал* обеспечивает механизм для индикации того, что в одной задаче потребовались данные другой задачи чтобы выполнить некую обработку (зависимость по данным)

Основные свойства модели «задача/канал»

- Эффективность
- Независимость отображения
- Модульность
- Детерминизм

Модель передачи сообщений

Возможно
- динамическое создание задач
- выполнение нескольких задач на одном процессоре
- выполнение различных программ различными задачами

НО в большинстве систем передачи сообщений:
- создается фиксированное число идентичных задач при запуске программы
- не поддерживается динамическое управление задачами во время выполнения программы
- используется модель программирования – SIMD: каждая задача выполняет одну и ту же программу, но функционирует на различных данных

Модель разделяемой общей памяти

- Задачи разделяют общее адресное пространство, чтение и запись в него выполняется асинхронно.
- Для управления доступом к общей памяти могут использоваться различные механизмы типа блокировок и семафоров
- Преимущество этой модели с точки зрения программиста - отсутствует понятие «собственника» данных, и, следовательно, нет необходимости явно указывать способ передачи данных от производителей потребителям (Это упрощает разработку программы)
- Понимание и управление локализацией данных более сложное
- Разработка детерминированных программ затруднена

Модели параллелизма данных

- Используется естественный параллелизм приложения или некой однотипной операции над элементами структуры данных например, «добавляется 2 к всем элементам этой таблицы» или «увеличивается оклад всех служащих со стажем свыше 5 лет»
- Программа с параллелизмом данных состоит из последовательности таких операций
- Компиляторы, использующие параллелизм данных, часто требуют, чтобы программист явно указал распределение данных по процессорам.

Модели параллельных алгоритмов

- Алгоритмы, использующие параллелизм данных (Data Parallelism)
- Алгоритмы в виде графа задач (Task Graph Model)
- Алгоритмы в виде пула задач (Work Pool Model)

Модели параллельных алгоритмов

- Менеджер-рабочий (Master-Slave Model)
- Конвейерные вычисления (Pipeline or Producer-Consumer)
- Гибридные модели

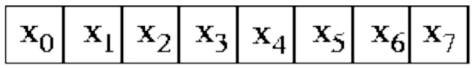
Примеры параллельных приложений

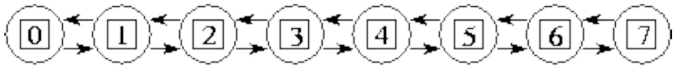
- Конечноразностные задачи
- Задачи поиска
- Задачи исследования параметров

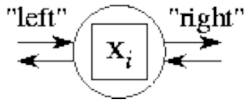
Пример конечноразностной задачи

Имеем: $X^{(0)}$ Найти: $X^{(T)}$

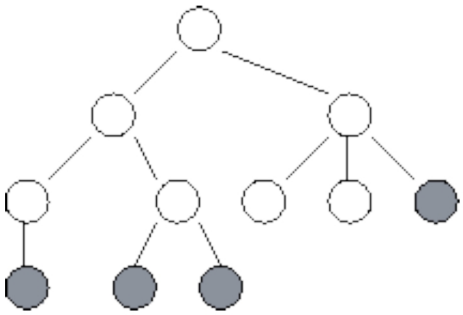
$$0 < i < N-1,\ 0 \le t < T\ :\ X_i^{(t+1)} = \frac{X_{i-1}^{(t)} + 2X_i^{(t)} + X_{i+1}^{(t)}}{4}.$$





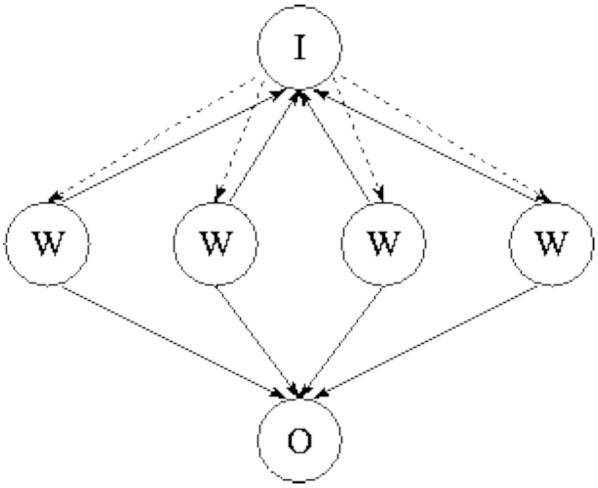


Пример задачи поиска



```
procedure search(A)
begin
  if (solution(A)) then
    score = eval(A)
    report solution and score
  else
    foreach child A(i) of A
      search(A(i))
    endfor
  endif
end
```

Пример задачи исследования параметров



Векторные и параллельные

```
real x(100), a(100), b(100)
do i=1, 100
   x(i) = real(i)**2 + 3.0
   a(i) = b(i) + x(i)
enddo
```

Параллельные

y = F(x) + G(x)

```
if(номер_процессора .eq. 1)
     then y1 = F(x)
else if(номер_процессора .eq. 2)
     then y2 = G(x)
endif (ждать завершения работы
         обоих процессоров)

if(номер_процессора .eq. 1)
     then y = y1 + y2
endif
```

Базовые операции параллелизма данных

- *Управление данными*
- *Операции над массивами*
- *Условные операции*
- *Операции приведения*
- *Операции сдвига*
- *Операции сканирования*
- *Операции пересылки данных*

FORTRAN+, MPP FORTRAN , HPF

Проблемы переноса последовательных программ на параллельные ЭВМ

```
float x[100], a[100];
register float *xx = x, *aa = a;
register int i;
for( i=0; i<100; i++)
   { *xx++ = *aa++ };
```

```
float x[100], a[100];
register int i;
for( i=0; i<100; i++)
   { x[i] = a[i]; }
```

```
real x(100), a(100)
do i=1, 100
   x(i) = a(i)
enddo
```

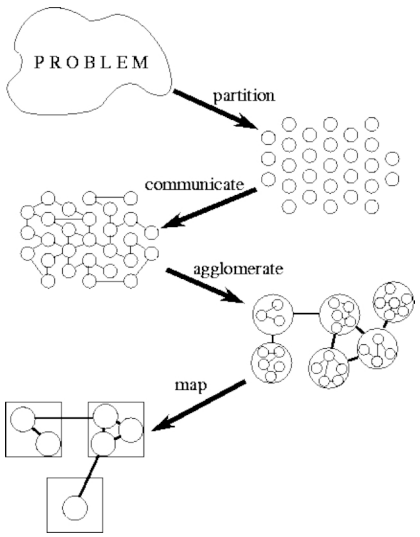
```
real x(100), a(100)
x = a ! векторное присваивание.
```

Методология проектирования параллельных приложений

Часть 1

Этапы процесса проектирования

- Декомпозиция *Partitioning*
- Взаимодействие *Communication*
- Интеграция *Agglomeration*
- Отображение *Mapping*



Основное внимание

- На первых двух этапах:
  - Параллелизм
  - Масштабируемость
- На третьем и четвертом этапах:
  - Локальность
  - другие проблемы эффективности

Декомпозиция *(Partitioning)*

- **Вычисление**, которое должно быть выполнено, и **данные**, необходимые для этого вычисления, делятся на **малые задачи**.
- Практические проблемы (число процессоров компьютера, на котором будет выполняться задача) игнорируются
- Внимание сосредоточено на обнаружении (выделении) возможностей **параллельного выполнения**

Взаимодействие *(Communication)*

- Определяются взаимодействия, требуемые для координации выполнения задачи
- Определяются соответствующие коммуникационные структуры и алгоритмы

Интеграция *(Agglomeration)*

- Задача и коммуникационные структуры, определенные на первых двух этапах проекта оцениваются в соответствии с требованиями эффективности и стоимости реализации.
- При необходимости задачи объединяются в большие задачи, чтобы улучшить быстродействие или уменьшить стоимость разработки.

Отображение *(Mapping)*

- Каждая задача назначается на процессор таким образом, чтобы попытаться удовлетворить двум противоречивым целям: максимизации использования процессора и уменьшения издержек связи
- Отображение может быть определено статически или выполнено динамически с помощью алгоритмов сбалансированной загрузки

Процесс проектирования

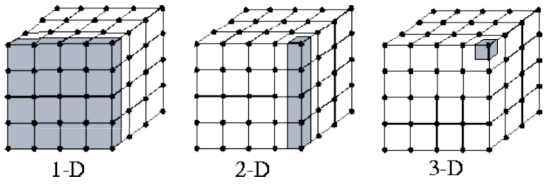
- Действие последовательное
- ***НО***, практически, это – **существенно параллельный процесс**
- Одновременно учитывается много аспектов
- Возможен, хотя и не желателен, перебор с возвратами
- Анализ части или всего проекта может потребовать изменений проектных решений, сделанных на предыдущих этапах

Декомпозиция (Partitioning)

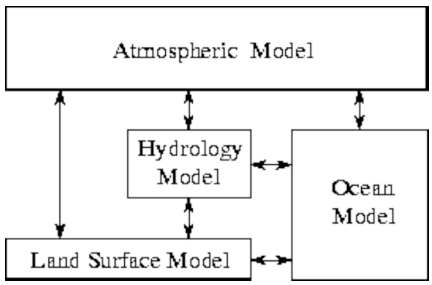
Этап 1
процесса проектирования

Типы декомпозиции

- Декомпозиция по данным



- Функциональная декомпозиция



Декомпозиция по данным

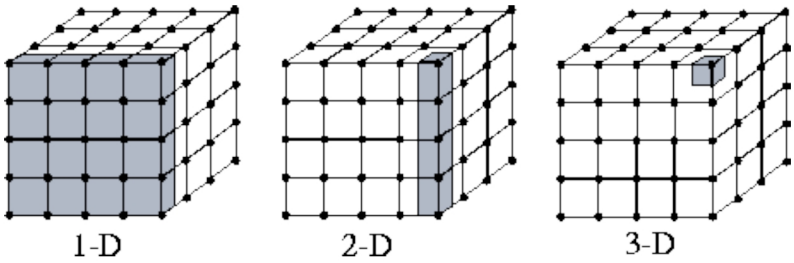
Данные, над которыми выполняется декомпозиция:

- Входные
- Выходные
- Промежуточные

Основное внимание -

- самой **большой** структуре данных
- структуре данных, к которой **обращаются** наиболее **часто**

Декомпозиция по данным на трехмерной решетке



- На ранних стадиях проектирования предпочтительна трехмерная декомпозиция
- Она предлагает самую большую гибкость

Функциональная декомпозиция

- Другой взгляд на проблему
- Альтернативный по отношению к декомпозиции по данным
- Дополняющий декомпозицию по данным

Функциональная декомпозиция

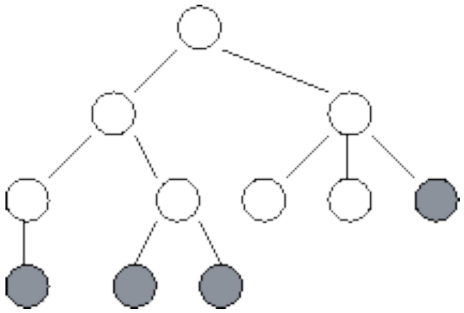
Сфокусируемся на вычислениях.

1. Делим вычисления на непересекающиеся задачи
2. Определяем данные, необходимые для каждой задачи

- Если требуемые для задачи данные также не пересекаются, то декомпозиция успешно определена
- Если данные задач существенно пересекаются – необходима декомпозиция по данным

Пример функциональной декомпозиции

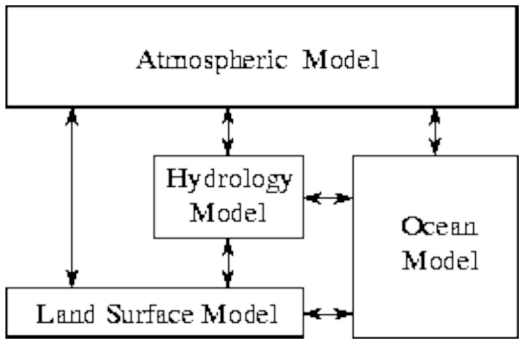
- Задача поиска



```
procedure search(A)
begin
  if (solution(A)) then
    score = eval(A)
    report solution and score
  else
    foreach child A(i) of A
      search(A(i))
    endfor
  endif
end
```

Функциональная декомпозиция

- Методика структурирования программы
- Уменьшение сложности всего проекта



Перечень контрольных вопросов

1. Определено ли в процессе декомпозиции на порядок большее количество задач, чем количество процессоров на параллельном компьютере?
- Если нет, то в последующих стадиях проектирования может не хватать гибкости.

Перечень контрольных вопросов

2. Используются ли избыточные вычисления?
   Предъявляются ли повышенные требования к объему памяти?

- Если да, то полученный алгоритм, возможно, не является масштабируемым при решении задач большой размерности

Перечень контрольных вопросов

3. Имеют ли задачи сравнимый размер?

- Если нет, могут возникнуть сложности при осуществлении равномерной загрузки процессоров.

Перечень контрольных вопросов

4. Число задач сопоставимо с размерностью задачи?

- Если дело обстоит не так, данный параллельный алгоритм, возможно, не сможет решать проблемы большой размерности даже при увеличении количества доступных процессоров.

- Идеально, если увеличение размерности задачи должно увеличивать
  в первую очередь **число** задач,
  а не **размер** индивидуальных задач.

Перечень контрольных вопросов

5. Определены ли альтернативные варианты декомпозиции?

- Рассмотрение вариантов на данном этапе может повысить гибкость последующих стадий проектирования. Не забудьте исследовать и территориальные, и функциональные варианты декомпозиции.

Взаимодействие
(Communication)

Этап 2
процесса проектирования

Зачем взаимодействие?

- Задачи выполняются параллельно, но они не могут, в большинстве случаев, выполняться независимо.
- Вычисление, которое нужно выполнить в одной задаче обычно требует данные, связанные с другой задачей.
- Чтобы процесс вычислений продолжался, данные должны передаваться между задачами.
- Этот поток информации определяется в коммуникационной фазе проектирования.

Основные шаги

- Определение структуры каналов,
  которые связывают напрямую или косвенно задачи(потребители), которым необходимы данные, и задачи(поставщики), которые формируют эти данные.
- Определение сообщений,
  которые будут передаваться по каналам
- Оптимизация затрат

Зависимость от технологии

В модели параллелизма данных:

- нет возможности явно создавать канал
- определяется распределение данных
- определяются операции над параллельными данными

Типы взаимодействий

- Локальные / Глобальные
- Структурные / Неструктурные
- Статические / Динамические
- Синхронные / Асинхронные

Локальное взаимодействие

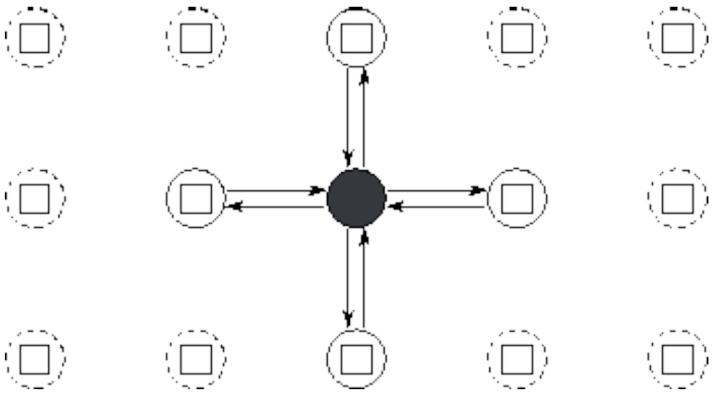
- Структура локального взаимодействия формируется, когда операция требует данных от малого числа других задач.
- Определяются каналы, которые связывают задачу, ответственную за выполнение операции (потребитель) с задачами, имеющими требуемые данные (производители).
- Определяются соответствующие операции посылки и получения в задачах производителях и потребителях, соответственно.

Пример
(метод конечных разностей Якоби)

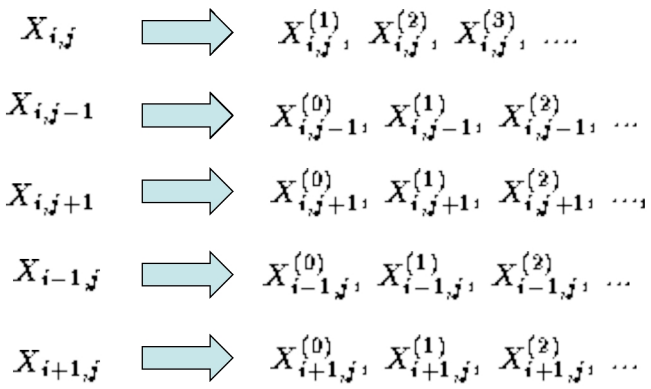
- Многомерная сетка итеративно модифицируется, заменяя значение в каждой точке некоторой функцией значений в малом, фиксированном числе соседних точек

$$X_{i,j}^{(t+1)} = \frac{4X_{i,j}^{(t)} + X_{i-1,j}^{(t)} + X_{i+1,j}^{(t)} + X_{i,j-1}^{(t)} + X_{i,j+1}^{(t)}}{8}.$$

Продолжение примера
(метод конечных разностей Якоби)



Продолжение примера (метод конечных разностей Якоби)



Логика каждой задачи

for *t=0* to *T-1*

send $X_{i,j}$ to each neighbor

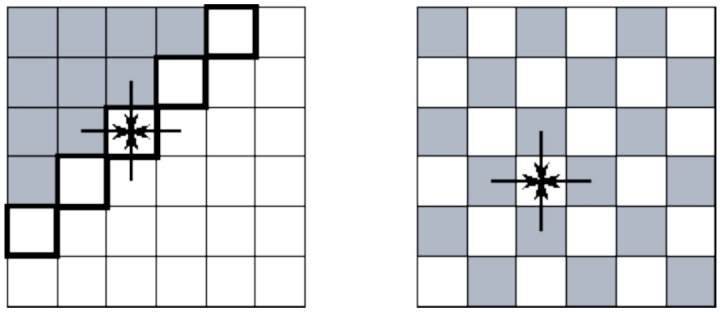
receive $X_{i-1,j}$ $X_{i,j-1}$ $X_{i+1,j}$ $X_{i,j+1}$

compute $X_{i,j}$ using Equation 2.1

endfor

Метод Гаусса-Зейделя

$$X_{i,j}^{(t+1)} = \frac{4X_{i,j}^{(t)} + X_{i-1,j}^{(t+1)} + X_{i+1,j}^{(t)} + X_{i,j-1}^{(t+1)} + X_{i,j+1}^{(t)}}{8}.$$

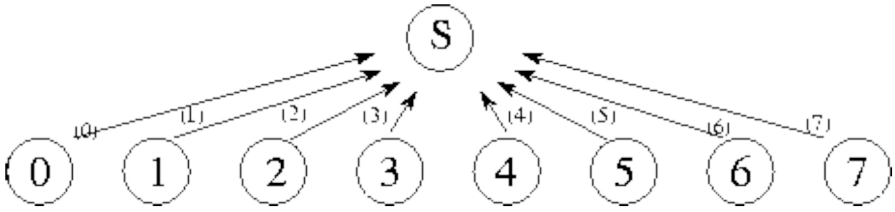


Глобальное взаимодействие

- Операция где должны участвовать много задач.
- При выполнении этих операций определение индивидуальной пары производителя / потребителя затруднено
- Может приводить к избыточным взаимодействиям
- Может ограничивать возможности параллельного выполнения.

Операция редукции

$$S = \sum_{i=0}^{N-1} X_i.$$

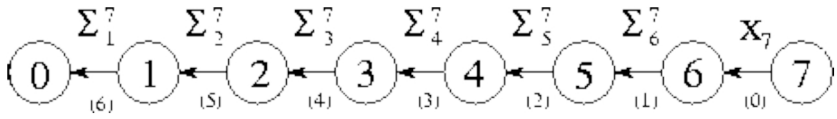


Проблемы локальных способов взаимодействия

- **Алгоритм - *централизованный***
  Вычисления и обмены не разделимы. Одна задача (задач-диспетчер) должна участвовать в каждой операции
- **Алгоритм - *последовательный***
  нет возможности одновременного выполнения операций вычислений и операций обмена

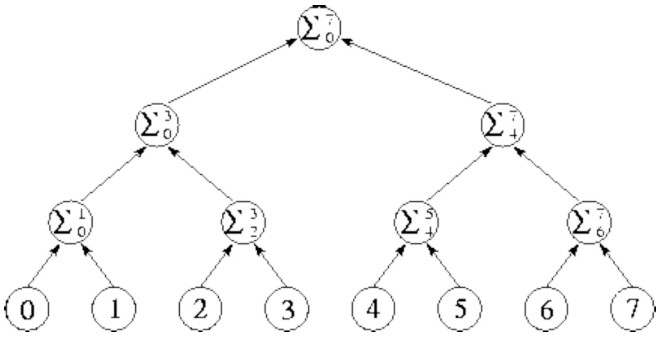
Распределение вычислений и взаимодействий

$$S_i = X_i + S_{i-1}.$$



Разделяй и властвуй

$$\sum_{i=0}^{2^n-1} = \sum_{i=0}^{2^{n-1}-1} + \sum_{i=2^{n-1}}^{2^n-1}$$

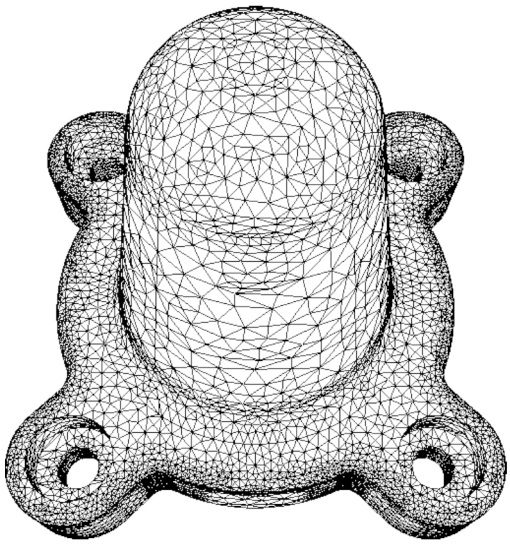


Полная сумма вычисляется за ***log N*** шагов

```
procedure divide_and_conquer
begin
   if base case then
      solve problem
   else
      partition problem into subproblems L and R
      solve problem L using divide-and-conquer
      solve problem R using divide-and-conquer
      combine solutions to problems L and R
   endif
end
```

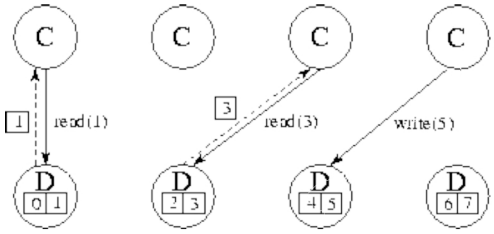
Algorithm 2.1 The divide-and-conquer strategy, a technique commonly used to uncover parallelism in algorithms. This procedure provides a recursive formulation of the strategy. If we are dealing with a primitive ("base case") problem, the problem is solved. Otherwise, the problem is divided into two subproblems. If these subproblems are independent, they can be solved concurrently.

Неструктурные и динамические связи



Асинхронные взаимодействия

- Задачи, которые обладают данными (производители), не способны определить, когда другие задачи (потребители) могут затребовать данные.
- Потребители должны явно запросить данные от производителей.



Стратегии асинхронных взаимодействий

- Структура данных распределена среди вычислительных задач (прерывание для выполнения ожидающих запросов)
- Распределенная структура данных инкапсулирована в набор задач, ответственных только за отклик на запросы чтения и записи
- Модель программирования с использованием разделяемой общей памяти

Перечень контрольных вопросов

1. Выполняют ли все задачи приблизительно одинаковое число операций обмена данными?

- Несбалансированные коммуникационные требования предполагают немасштабируемую конструкцию.
- Можно ли операции обмена перераспределить более равномерно?
- Если структура данных, к которой часто обращаются, инкапсулирована в одной задаче, рассмотрите возможности распределения или репликации этой структуры данных.

Перечень контрольных вопросов

2. Взаимодействует ли каждая задача только с малым числом соседей?

- Если каждая задача должна взаимодействовать с большим количеством других задач, оцените возможность формулировки такого глобального взаимодействия в терминах локальной коммуникационной структуры, как это было сделано в алгоритме парных взаимодействий

Перечень контрольных вопросов

3. Способны ли операции обмена выполняться одновременно?

- Если нет, то вероятно, что ваш алгоритм не является эффективным и масштабируемым.
- Попробуйте использовать метод "разделяй и властвуй", чтобы выявить параллелизм

Перечень контрольных вопросов

4. Возможно ли одновременное выполнение процесса вычислений, распределенного по различным задачам?

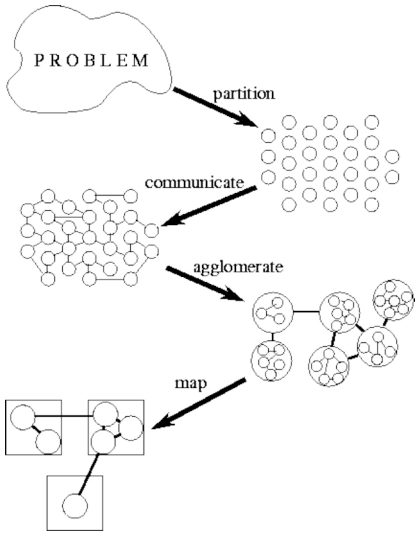
- Если нет, то весьма вероятно, что ваш алгоритм не является эффективным и масштабируемым.
- Можно ли переупорядочить операции вычисления и обмена.
- Может быть желательно переформулировать задачу

Методология проектирования параллельных приложений

Часть 2

Этапы процесса проектирования

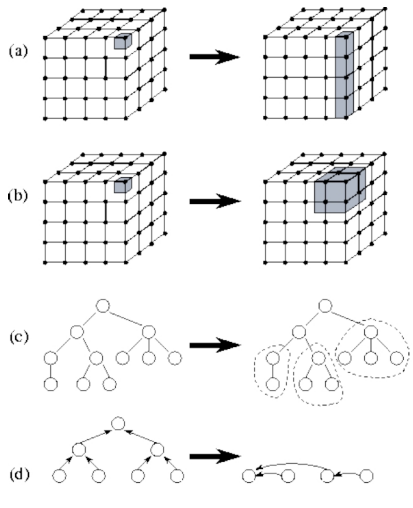
- Декомпозиция *Partitioning*
- Взаимодействие *Communication*
- Интеграция *Agglomeration*
- Отображение *Mapping*



Интеграция
Agglomeration

Этап 3
процесса проектирования

Примеры укрупнения



- Уменьшение размерности декомпозиции
- Объединение смежных задач - повышение степени детализации
- Объединение поддеревьев в структуре «разделяй и властвуй»
- Объединение узлов в дереве

Цели интеграции

- Уменьшение коммуникационных издержек
- Сохранение гибкости
- Сокращение стоимости разработки ПО

Критические проблемы этапа

- Затраты на коммуникацию
  - Количество пересылок
  - Объем пересылаемых данных
- Затраты на создание задач

Уменьшение коммуникационных издержек

Основные подходы:

- Укрупнение элементов разбиения
- Репликация вычислений
- Репликация данных
- Устранение лишних связей

Укрупнение элементов разбиения

- Уменьшение количества и объема коммуникаций путем объединения отдельных задач в одну

- Критерий - отношение поверхности к объему:
- Затраты на связь ~ поверхности подобласти
- Вычисления ~ объему подобласти

Пример двумерной задачи

"поверхность" ~ N

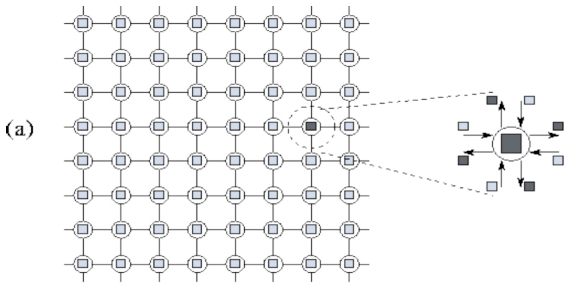
"объем" ~ N^2

(N - размер задачи)

Следовательно,

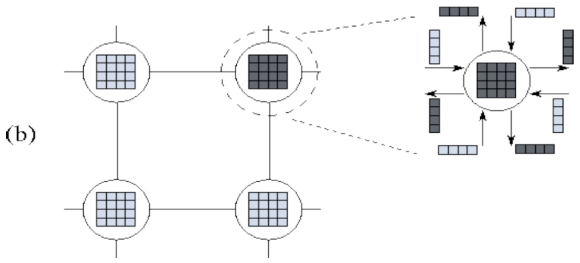
- **количество взаимодействий**, выполняемых для вычислительной единицы (коэффициент обмен данными / вычисления) **уменьшается при увеличении размера задачи**

Пример двумерной проблемы



сетка 8×8

(a) **8×8=64 задач**
64×4=256 связей
256 значений данных

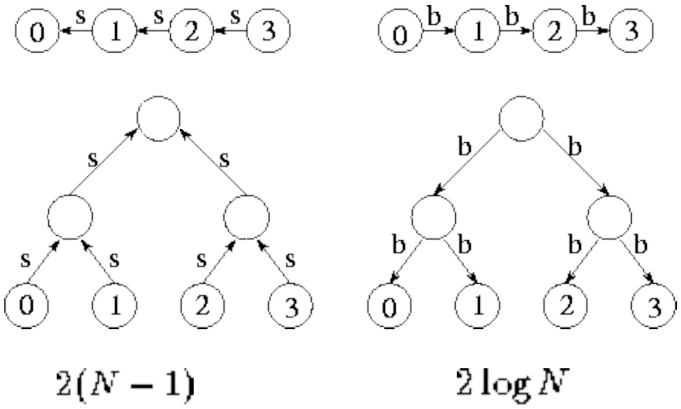


(b) **2×2=4 задачи**
4×4=16 связей
16×4=64 значений данных

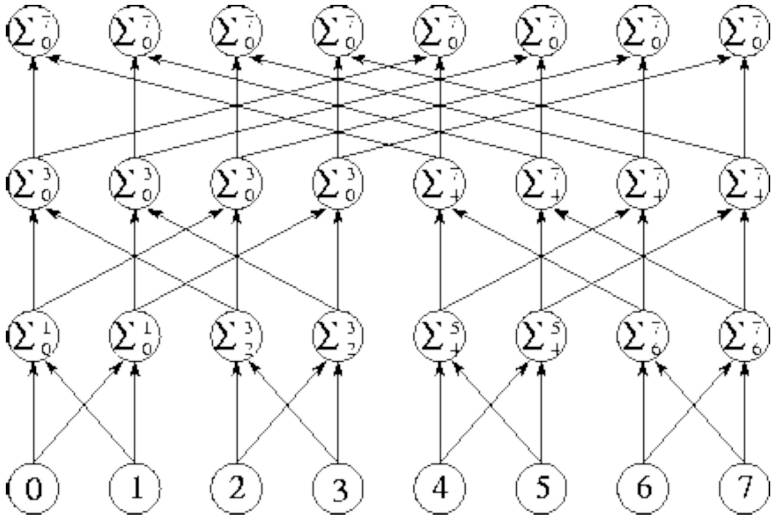
Следствие эффектов «поверхность / объем»

- Декомпозиции с большей размерностью являются обычно наиболее эффективными (при прочих равных условиях), потому что они уменьшают поверхностную область (связь), требуемую для данного объема (вычисление).
- Следовательно, более эффективно укрупнять элементы, интегрируя задачи по всем размерностям, нежели уменьшать размерность декомпозиции.

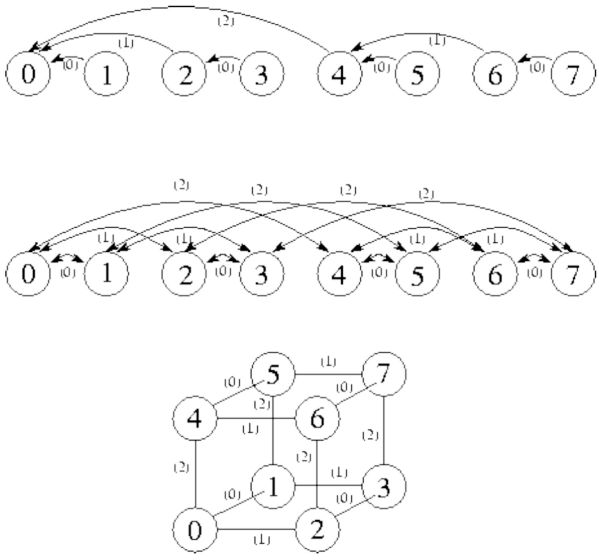
Репликация вычислений



Репликация вычислений



Устранение лишних связей



Сохранение гибкости

- Число задач превышает количество доступных процессоров
- Оптимальное число задач определяется комбинацией аналитического анализа модели и эмпирических исследований
- Степень детализации – параметр этапа компиляции или этапа выполнения
- Нет ограничений на число создаваемых задач

Сокращение стоимости разработки ПО

- Относительные издержки разработки различных стратегий декомпозиции
- Распределение данных, используемое другими компонентами большой системы

Контрольные вопросы этапа

1. Издержки связи уменьшены? Локализация увеличилась?

- Если нет, исследуйте алгоритм, чтобы определить, возможно ли этого достигнуть, используя альтернативную стратегию скопления.

Контрольные вопросы этапа

2. Получена ли выгода от репликации вычислений?
3. Если реплицируются данные, не ухудшает ли это масштабируемость алгоритма?
4. Получены ли задачи с приблизительно одинаковыми вычислительными и коммуникационными потребностями?

Контрольные вопросы этапа

5. Число задач все еще соотносится с размером задачи?
6. Если интеграция уменьшила степень параллелизма выполнения задачи, проверьте, осталось ли достаточно параллелизма для используемого и предполагаемого компьютера?

Контрольные вопросы этапа

7. Может ли и далее уменьшить число задач, без появления несбалансированности загрузки, увеличения издержек разработки ПО, или уменьшения масштабируемости?

- При прочих равных условиях, алгоритмы, которые создают меньшее количество больших задач, зачастую проще и эффективнее тех, которые создают много мелких задач.

Контрольные вопросы этапа

8. Если распараллеливается существующая последовательная программа, определена ли стоимость внесения изменений в последовательный код?
- Если эти издержки высоки, надо рассмотреть альтернативные стратегии интеграции, увеличивающие возможности многократного использования кода.
- Используйте методы оценки эффективности, чтобы оценить компромиссы стоимости.

Отображение
Mapping

Этап 4
процесса проектирования

Основная цель этапа

- Минимизация общего времени выполнения параллельной задачи

Сложности достижения цели:

- Ограниченность ресурсов может лимитировать число задач, размещаемых на одном процессоре
- Проблема отображения является NP-полной задачей

Стратегии достижения цели

- Задачи, которые могут выполняться одновременно, размещаются на различных процессорах, расширяя, таким образом, параллелизм.

- Задачи, которые часто взаимодействуют, размещаются на одном и том же процессоре, увеличивая, таким образом, локализацию.

Стратегия для

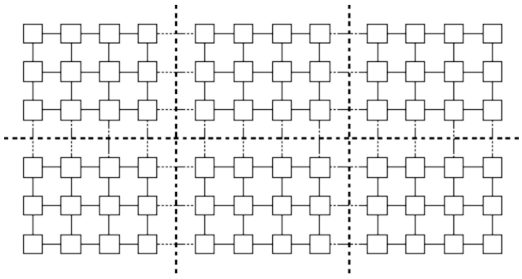
- алгоритмов с декомпозицией по данным, которые отличаются фиксированным числом задач одинакового размера и локальными и глобальными коммуникациями, имеющими структурную определённость.

- Прямое (простое) отображение является эффективным.

Прямое (простое) отображение

- Задачи отображаются таким способом, чтобы минимизировать межпроцессорное взаимодействие
- Возможно укрупнение задач, отображенных на один и тот же процессор, если это еще не было сделано, с тем, чтобы получить *P* крупно модульных задач, по одной на каждый процессор.

Прямое (простое) отображение



- Каждому процессору сопоставляется одинаковый объем вычислений
- Минимизируются межпроцессорные взаимодействия

Стратегия для

- алгоритмов, основанных на декомпозиции по данным с переменным объемом вычислений у задач и/или неструктурированным характером взаимодействий.
- Эффективные стратегии интеграции и отображения могут быть не очевидны для программиста

Алгоритмы сбалансированной загрузки

- Используют эффективные стратегии укрупнения и отображения, используя эвристические методы.
- Необходимо сопоставлять время, требуемое для работы этих алгоритмов, с уменьшением общего времени выполнения.
- Вероятностные методы сбалансированной загрузки имеют более низкие накладные расходы, чем, методы, которые опираются на знание структуры приложения.

Стратегия для

- приложений, в которых либо число задач, либо объем вычислений или коммуникаций, приходящийся на задачу, изменяется динамически во время выполнения программы.

- Динамическая стратегия сбалансированной загрузки

Динамическая стратегия сбалансированной загрузки

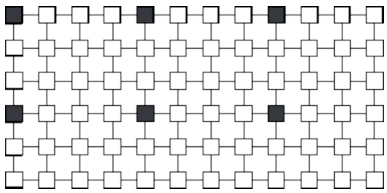
- Алгоритм сбалансированной загрузки выполняется периодически, переопределяя укрупнение и отображение.
- Поскольку во время выполнения программы определение сбалансированности загрузки должно выполняться многократно, в данном случае предпочтительны локальные алгоритмы, которые не требуют глобальных знаний относительно состояния вычислений.

Стратегия для

- алгоритмов, основанных на функциональной декомпозиции. Большое количество короткоживущих задач, которые координируют с другими задачами только в начале и в конце своего выполнения.
- Алгоритмы диспетчеризации, которые распределяют задачи по процессорам, которые являются неактивными, или в скором времени станут таковыми.

Алгоритмы сбалансированной загрузки

- Рекурсивные методы биекции (деления пополам)
- Локальные алгоритмы
- Вероятностные методы
- Методы циклического отображения

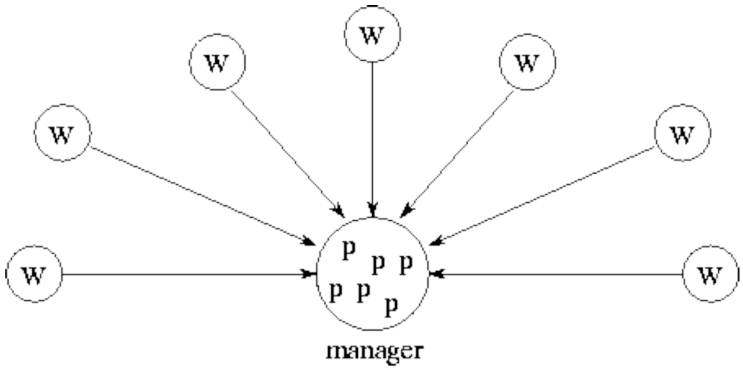


Алгоритм планировщика задач

- Используется, когда функциональная декомпозиция приводит к задачам со слабыми требованиями к локальности.
- Централизованный или распределенный пул задач, в который помещаются новые задачи и откуда они выбираются для распределения по процессорам.

Варианты

«менеджер-рабочий»



Варианты

- Иерархический «менеджер-рабочий»
- Децентрализованные схемы

- Проблема *обнаружения завершения*

Перечень контрольных вопросов

1. Рассмотрен ли алгоритм, основанный на динамическом создании и удалении задач?
2. При использовании централизованной сбалансированной загрузки, не является ли менеджер узким местом?

- Затраты связи можно уменьшить, передавая менеджеру указатели на задачи, а не собственно задачи.

Перечень контрольных вопросов

3. При использовании динамических алгоритмов сбалансированной загрузки, оценены ли относительные затраты различных стратегий?

- Убедитесь, что учитываются затраты на выполнение алгоритма балансировки.
- Вероятностные или циклические схемы отображения просты и нужно всегда их рассматривать, потому что они могут избежать потребности в повторных этапов балансировки.

Перечень контрольных вопросов

4. При использовании вероятностных или циклических методов, имеется ли достаточно большое число задач, чтобы гарантировать корректную сбалансированность?

- Как правило, требуется по крайней мере в десять раз больше задач, чем количество процессоров, имеющихся в наличии.

Методология проектирования параллельных приложений

Часть 3

Примеры разработки

Моделирование атмосферы

Пример задачи удобной для распараллеливания с помощью декомпозиции по данным

Conservation of momentum:

$$\frac{du}{dt} - \left(f + u\frac{\tan\phi}{a}\right)v = -\frac{1}{a\cos\phi}\frac{1}{\rho}\frac{\partial p}{\partial\lambda} + F_\lambda$$

$$\frac{dv}{dt} + \left(f + u\frac{\tan\phi}{a}\right)u = -\frac{1}{\rho a}\frac{\partial p}{\partial\phi} + F_\phi$$

Hydrostatic approximation:

$$g = -\frac{1}{\rho}\frac{\partial p}{\partial z}$$

Conservation of mass:

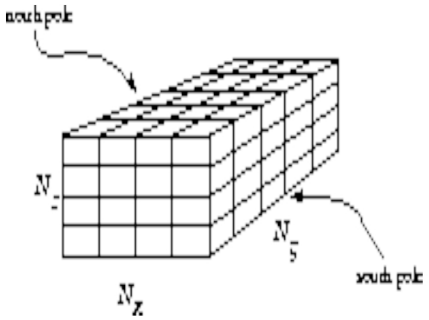
$$\frac{\partial\rho}{\partial t} = -\frac{1}{a\cos\phi}\left(\frac{\partial}{\partial\lambda}(\rho u) + \frac{\partial}{\partial\phi}(\rho v\cos\phi)\right) - \frac{\partial}{\partial z}(\rho w)$$

Conservation of energy:

$$C_p\frac{dT}{dt} - \frac{1}{\rho}\frac{dp}{dt} = Q$$

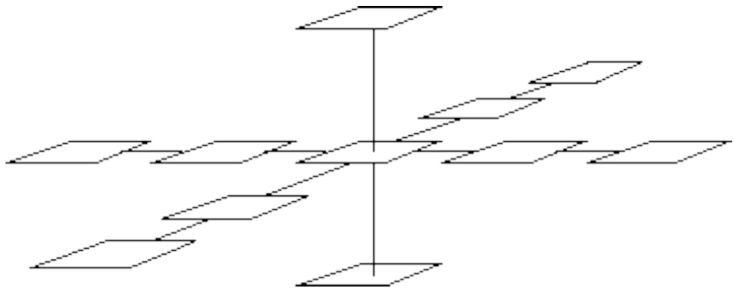
State equation (atmosphere):

$$p = \rho RT$$



Состояние атмосферы представляется на трехмерной решетке

Для выполнения итерационного пересчета значений параметров состояния атмосферы в узлах решетки используются 9-точечный шаблон для моделирования горизонтального перемещений и 3-точечный – для вертикальных.

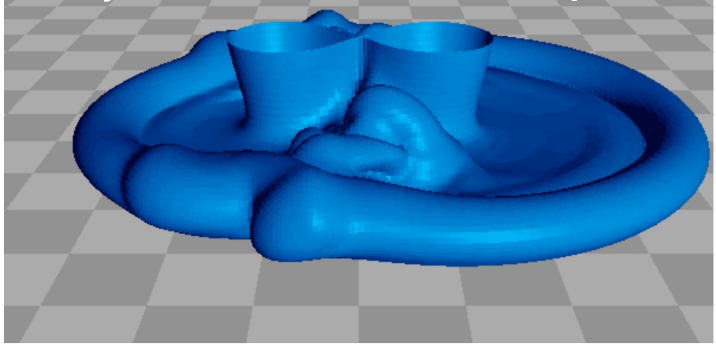


Физические процессы

- Излучение (солнечное)
- Конвекция
- Осадки

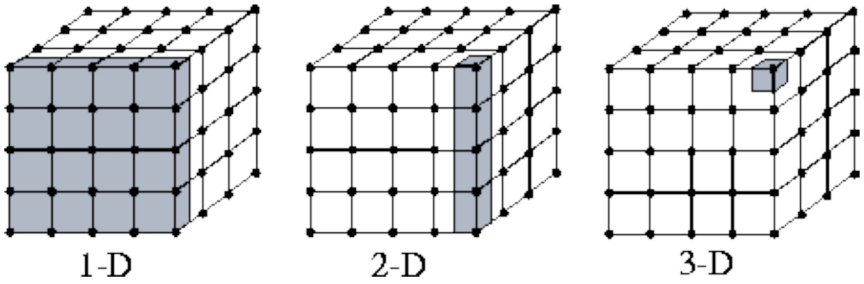
При вычислениях используются данные, соответствующие одному столбцу.

Поверхность равных значений возможной температуры двух нисходящих потоков грозы



Область столкновения содержит поля воздушных потоков, которые опасны для приземляющегося самолета.
Размер стороны серого квадрата – 1 километр.
Область моделирования – 16×18 км с разрешением в 50 м.

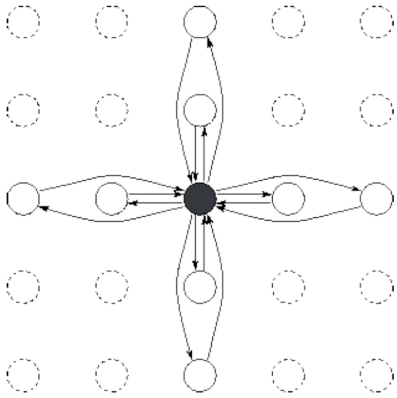
Декомпозиция



- Используется трехмерная декомпозиция

Коммуникации

1. Шаблона конечных разностей (локальные)



Коммуникации

2. Глобальных вычислений

$$Total\ Mass = \sum_{i=0}^{N_x-1} \sum_{j=0}^{N_y-1} \sum_{k=0}^{N_z-1} M_{ijk},$$

3. Вычислений, связанных с физическими процессами

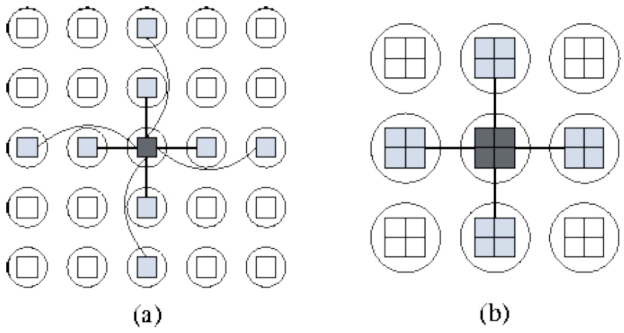
$$\begin{aligned} \mathrm{TCS}_k &= \prod_{i=1}^{k} (1 - \mathrm{cld}_i) TCS_1 \\ &= TCS_{k-1}(1 - \mathrm{cld}_k), \end{aligned}$$

Анализ этапа № 2

- Локальные коммуникации – распределенные, могут выполняться параллельно.
- Для выполнения глобальных вычислений может использоваться рассредоточенный алгоритм.
- Глобальные вычисления можно выполнять не на каждом шаге.
- Вычисления физических процессов являются узким местом данного этапа.

Интеграция

1. Небольшая интеграция (от одной к четырем точкам сетки на задачу) уменьшает коммуникационные требования девяти точечного шаблона с восьми до четырех сообщений на задачу для одного шага итерации.



Интеграция

2. Коммуникационные требования в горизонтальном измерении относительно небольшие: четыре сообщения по восемь значений данных.
   В вертикальном измерении необходимы коммуникации не только для шаблона конечных разностей (2 сообщения по 2 значения данных), но и для вычислений "физики" (30 сообщений).
   Этих сообщений можно избежать, объединяя задачи в пределах каждого вертикального столбца.

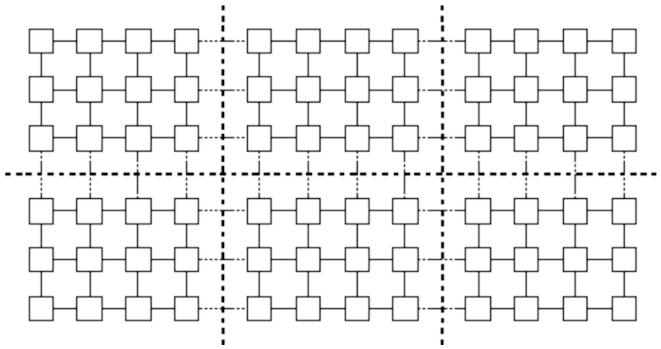
Интеграция

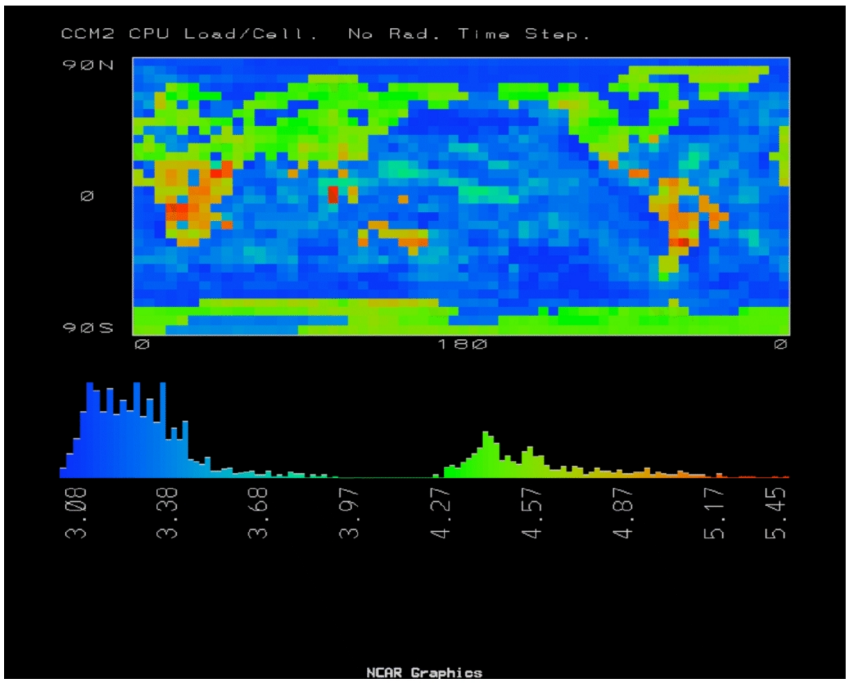
3. Вертикальная интеграция приветствуется и с точки зрения разработки программного обеспечения.
   Двумерная горизонтальная декомпозиция позволяет повторно использовать существующий последовательный код "физики" в параллельной программе без модификации.

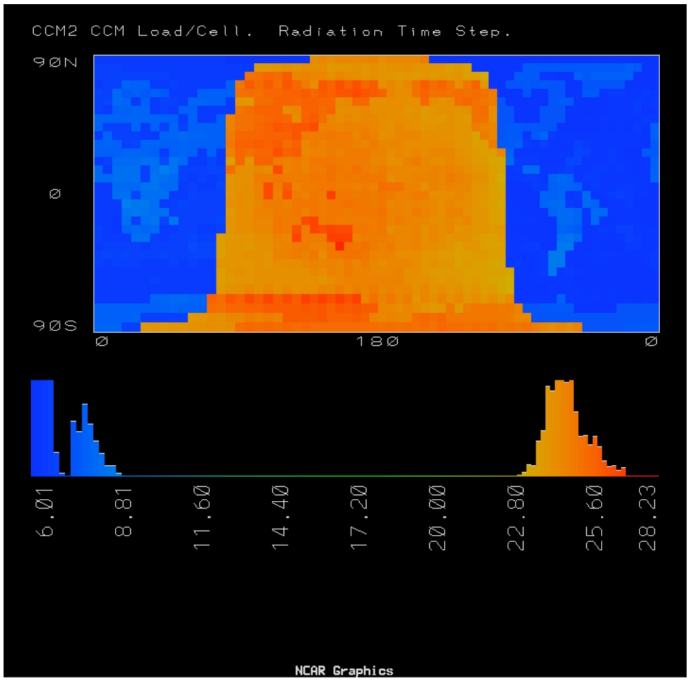
- Окончательно $N_x \times N_y/4$ задач.
  Приблизительно 10^3 – 10^5.

Отображение

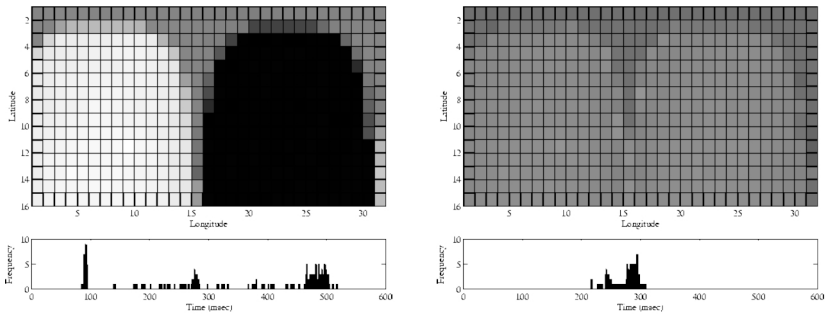
- Хорошая сбалансированность задач позволяет применить простую стратегию отображения







Использование циклического отображения

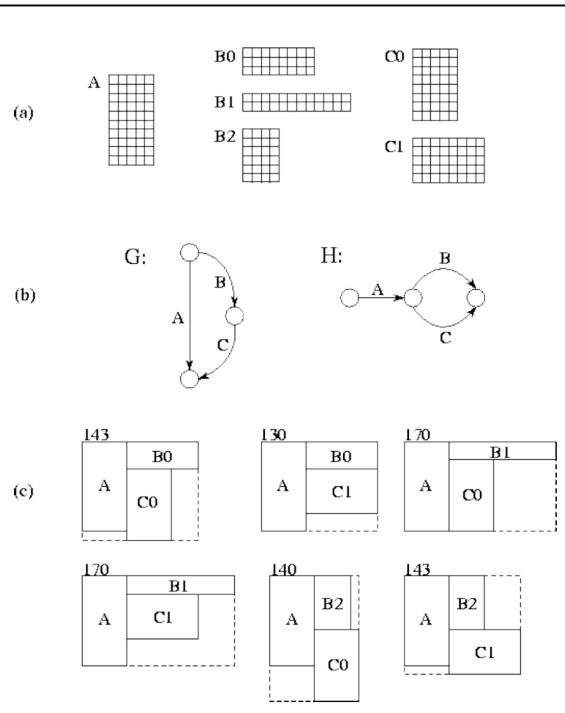


- На каждом процессоре располагаются задачи с западного и восточного,
  с северного и южного полушарий.

Оптимизация топологической структуры

Пример 2
разработки параллельного приложения

- Набор компонентов с различными вариантами исполнения
- Ограничения по размещению (смежность элементов)
- Возможные варианты топологии



Общее количество возможных конфигураций

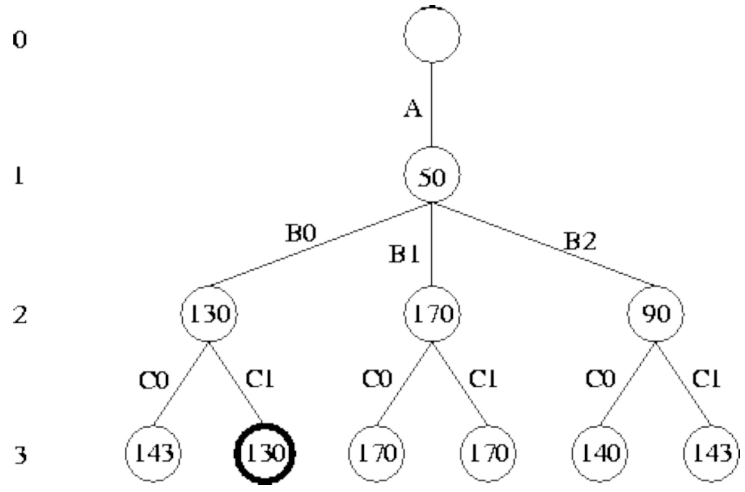
$$\prod_{i=0}^{N} I(c_i)$$

где

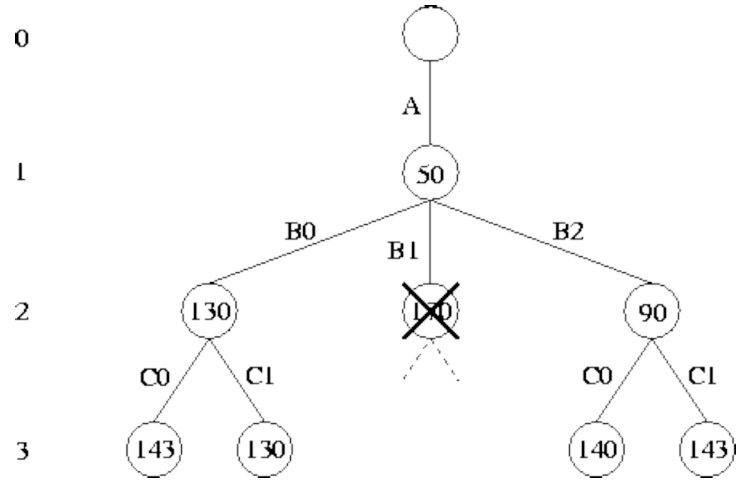
N – число элементов

$I(c_i)$ – количество реализаций элемента c_i

Полное дерево решения задачи размещения



Метод ветвей и границ




```
procedure b&b_search(A )
begin
  A_min = ∞
  b&b_search_1(A )
end

procedure b&b_search_1(A )
begin
  score = eval(A )
  if(score < A_min) then
    if (leaf(A )) then
      A_min = score
      report solution and score
    else
      foreach child A_i of A
        b&b_search_1(A_i)
      endfor
    endif
  endif
end
```

Декомпозиция

- Нет очевидной структуры данных
- Функциональное разложение, в котором каждый узел дерева поиска анализируется отдельной задачей
- Процесс поиска на дереве имеет тенденцию исполняться в режиме *«перебор вширь».*

Декомпозиция

- Новые задачи создаются на фронте процесса вычислений по мере продвижения его вниз по дереву поиска
- Только задачи на фронте процесса могут выполняться одновременно
- Проблема получения текущего минимального значения площади размещения ***всеми*** задачами
- Начальное решение – инкапсуляция текущего минимального значения площади размещения в отдельной задаче

Анализ этапа 1

- Стратегия исследования *«перебор вширь»* существенно снижает производительность,
задерживая нахождение решения в узлах и, следовательно,
сокращая количество отсечений ветвей,
приводя к значительным избыточным вычислениям.

Коммуникации

- Коммуникации требуются в процессе выполнения задачи для получения и обновления текущего минимального значения площади размещения A_{min}
- **Подход 1:**
инкапсуляция A_{min} в одной задаче
- простота, эффективен, если коммуникационные издержки невелики
- плохая масштабируемость

Коммуникации

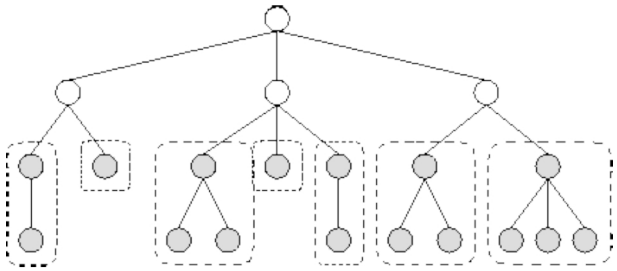
Усовершенствования:

- проверка A_{min} не на каждом уровне
- разбиение полного дерева на поддеревья с собственным подменеджером A_{min} и периодическими обменами информацией между подменеджерами. Подменеджеры могут выполнять широковещательную рассылку, когда они обнаруживают лучшие решения.

Интеграция (решаемые проблемы)

- Стоимость создания большого количества небольших задач
  Решение:
  переключение на последовательный поиск начиная с некоторого уровня
- Порядок обхода дерева не детерминирован
  Решение:
  использование планировщика задач

Интеграция



- Переключение на последовательный поиск на уровне 2 в дереве поиска
- Задача создается для каждого поддерева с корнем на уровне два

Отображение

- Метод планировщика задач
- Схема «менеджер-рабочий»
- Иерархический «менеджер-рабочий»

Анализ эффективности параллельных алгоритмов

Лекция

- Математическая модель характеристик параллельной программы
- Использование модели для сравнения эффективности различных программ
- Оценка масштабируемости
- Идентификация узких мест и других неэффективных решений
- Все это до реализации

Эффективность параллельной программы

- Время выполнения
- Масштабируемость процесса вычислений
- Устройства генерации, хранения данных, передачи по сети, чтения/записи на диск, передачи данных между различными этапами вычислений
- Стоимость различных стадий проектирования

Система показателей оценки характеристик

- Время выполнения
- Эффективность распараллеливания
- Требования по памяти
- Производительность
- Латентность
- Скорость ввода/вывода
- Переносимость
- Масштабируемость

Система показателей оценки характеристик

- Стоимость разработки
- Стоимость реализации
- Стоимость верификации
- Возможность повторного использования
- Аппаратные требования
- Стоимость аппаратуры
- Стоимость эксплуатации

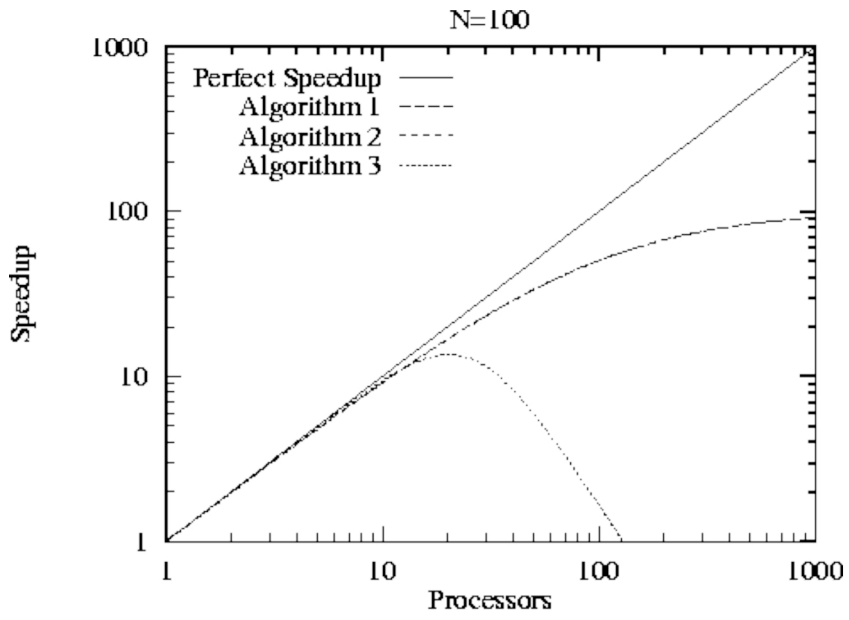
Экстраполяция по результатам наблюдения

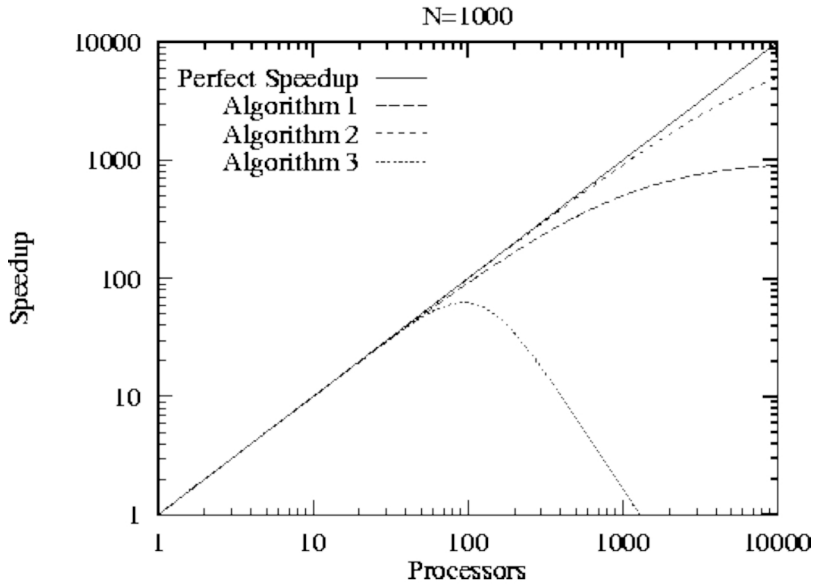
- Параллельный алгоритм *X* обеспечил коэффициент ускорения вычислений 10.8 для компьютера с 12 процессорами при размере задачи *N=100* .
- Хорошо это или плохо?
- А на 1000 процессорах?
- Для *N=10* или *N=1000* ?
- А если затраты на коммуникацию возрастут в 10 раз?

Три примера

- Модель определяет время выполнения *T* как функцию от числа процессов *P* и размера задачи *N*.
- Для последовательного алгоритма $T=N+N^2$

1. $T=N+N^2/P$
2. $T=(N+N^2)/P+100$
3. $T=(N+N^2)/P+0.6P^2$





Асимптотический анализ

- Время работы алгоритма ~ $O(N \log N)$ на $O(N)$ процессорах
- Это означает, что существует константа c и минимальный размер задачи N_0 такие, что для всех $N>N_0$ временные затраты $\leq c\, N \log N$ на N процессорах.

Недостатки асимптотического анализа

- Игнорируются малые значения *N* и *P*.
  T ~ N log N T=10N+N log N (N < 1024)
- *1000N log N vs. $10N^2$ (лучше при N<996)*
- Нет привязки к конкретному параллельному компьютеру (архитектуре)
- Поэтому – использовать осторожно!

Модель производительности

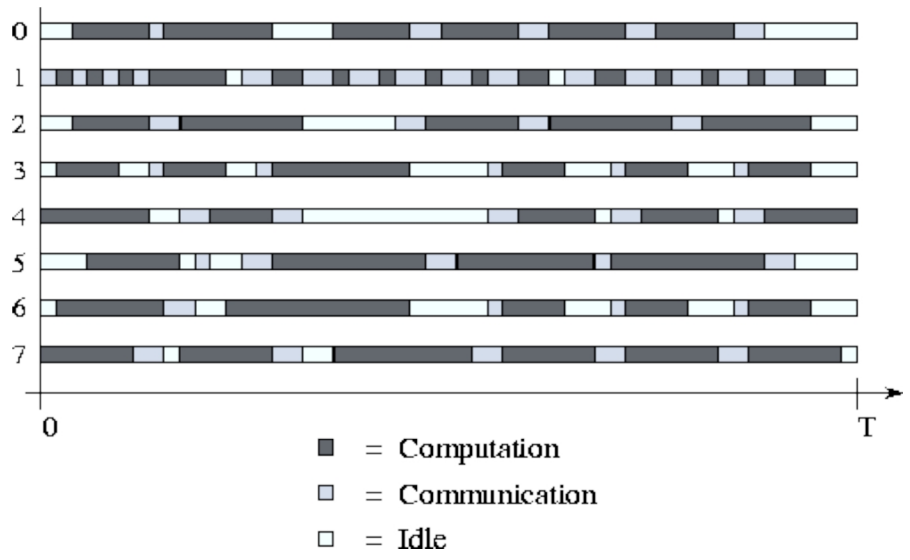
$$T = f(N, P, U, \ldots).$$

$$T^i_{\text{comp}} \qquad T^i_{\text{comm}} \qquad T^i_{\text{idle}}$$

$$T = T^j_{\text{comp}} + T^j_{\text{comm}} + T^j_{\text{idle}},$$

$$T = \frac{1}{P}(T_{\text{comp}} + T_{\text{comm}} + T_{\text{idle}})$$

$$= \frac{1}{P}\left(\sum_{i=0}^{P-1} T^i_{\text{comp}} + \sum_{i=0}^{P-1} T^i_{\text{comm}} + \sum_{i=0}^{P-1} T^i_{\text{idle}}\right).$$



Снижение сложности модели

- Используется идеализированный мультикомпьютер
- Масштабный анализ, т.е. определение незначащих составляющих задачи.
- Эмпирические исследования для уточнения простой модели vs. усложнение модели

Время вычислений

T_{comp}^{i}

Чаще всего зависит от

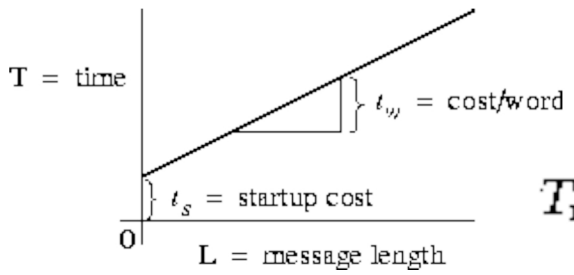
- размера задачи *N*
- характеристик процессоров
- параметров оперативной памяти

Время коммуникаций

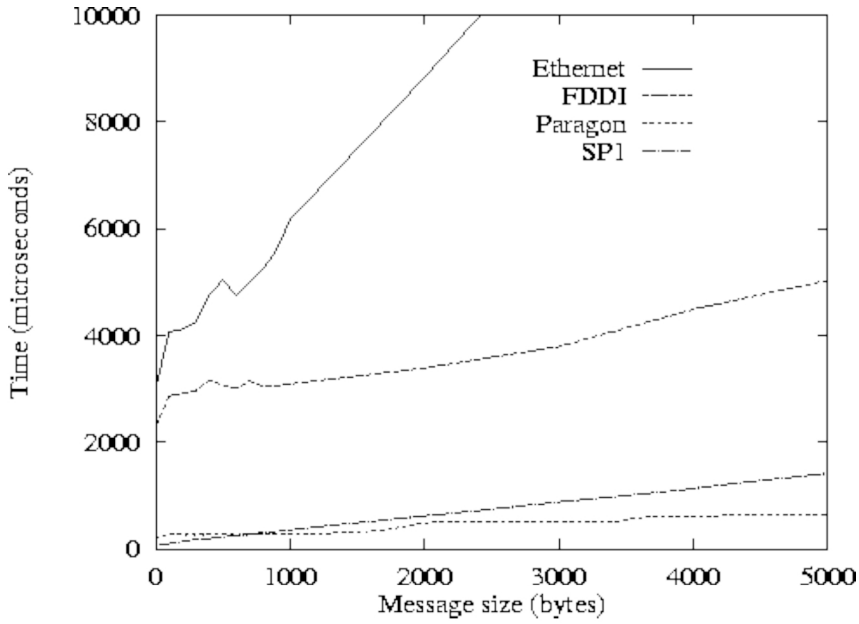
T_{comm}

Различают два типа коммуникаций:

- межпроцессорные (*interprocessor)*
- внутри процессорные (*intraprocessor*)

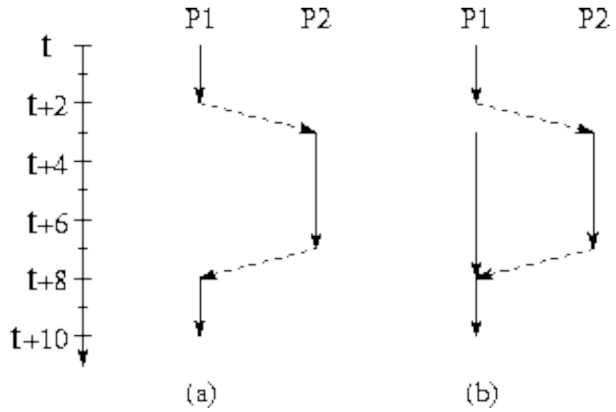


$$T_{msg} = t_s + t_w L.$$



T_{idle}

Idle Time



Параллельный конечно-разностный алгоритм

$N \times N \times Z$ $\qquad\qquad$ $N \times (N/P) \times Z$

$$T_{\text{comp}} = t_c N^2 Z,$$
$$T_{\text{comm}} = 2P\left(t_s + t_w 2NZ\right).$$

$$\begin{aligned} T_{\text{1d finite diff.}} &= \frac{T_{\text{comp}} + T_{\text{comm}}}{P} \\ &= t_c \frac{N^2 Z}{P} + t_s 2 + t_w 4NZ. \end{aligned}$$

Эффективность и ускорение

- Относительная эффективность:

$$E_{\text{relative}} = \frac{T_1}{PT_P},$$

- Относительное ускорение:

$$S_{\text{relative}} = PE,$$

Конечно-разностный алгоритм

$$T_1 = t_c N^2 Z \qquad E = \frac{t_c N^2 Z}{t_c N^2 Z + t_s 2P + t_w 4NZP}$$

- Эффективность уменьшается с возрастанием P, t_s *u* t_w.
- Эффективность возрастает с возрастанием N, Z, t_c.
- Время выполнения уменьшается с увеличением P.
- Время выполнения возрастает с увеличением N, Z, t_c, t_s *u* t_w

Модель используется для ответа на следующие вопросы:

- Удовлетворяет ли алгоритм предъявляемым требованиям? (по времени, по памяти, и т.п.)
- Насколько он адаптируем? Как он себя поведет при увеличении N, P? Как он зависит от t_s *u* t_w ?
- Каковы параметры алгоритма в сравнении с другими аналогичными ?

Анализ масштабируемости при фиксированных параметрах

- Как изменятся *T* и *E* при увеличении *P* если размер задачи и параметры компьютера фиксированы?
- Насколько быстро я смогу решить задачу *A* на компьютере *X* ?
- Какое число процессоров необходимо для увеличения эффективности на 50% ?
- *T* и *E* анализируются совместно !

Анализ масштабируемости задачи

- Мы рассматриваем не то, как *E* изменяется с изменением *P* , а то, как количество выполняемых вычислений должно соотноситься с *P* чтобы *E* не изменилось.
- Эта функция от *N* называется функцией изоэффективности алгоритма
- Характеризует поведение алгоритма
- Алгоритм с ФИЭ $O(P)$ имеет отличную масштабируемость $O(P^2)$ – хуже

Для получения функции изоэффективности используется следующее соотношение

$$T_1 = E\left(T_{\text{comp}} + T_{\text{comm}} + T_{\text{idle}}\right).$$

Изоэффективность конечно-разностного алгоритма

$N \times N \times Z$

$$t_c N^2 Z \approx E(t_c N^2 Z + t_s 2P + t_w 4NZP).$$

N=P

$$t_c Z \approx E\left(t_c Z + t_s \frac{2}{P} + t_w 4Z\right).$$

$$\mathcal{O}(P^2)$$

Изоэффективность конечно-разностного алгоритма

$(N/\sqrt{P}) \times (N/\sqrt{P}) \times Z \qquad 2(N/\sqrt{P})Z$

$$E = \frac{t_c N^2 Z}{t_c N^2 Z + t_s 4P + t_w 8NZ\sqrt{P}}.$$

$$t_c N^2 Z \approx E\left(t_c N^2 Z + t_s 4P + t_w 8NZ\sqrt{P}\right).$$

$N = \sqrt{P}$

$$t_c Z \approx E\left(t_c Z + t_s 4 + t_w 8Z\right).$$

$$\mathcal{O}(P)$$

Библиотека MPI
Message Passing Interface

Основные свойства

- MPI - *Message Passing Interface*, интерфейс передачи сообщений
- Стандарт MPI 1.1
- Языки программирования:
  - FORTRAN
  - C
- Более 120 функций
- SPMD-модель параллельного программирования

Основные свойства

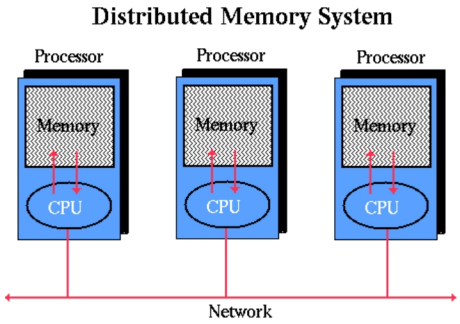
- Наличие групп процессов (безопасность сообщений), топологий процессов
- Структурирование передаваемого сообщения, типизация, гетерогенность
- Режимы: normal (blocking and non-blocking), synchronous, ready (to allow access to fast protocols), buffered
- Большое разнообразие коллективных операций

Чего нет в MPI

- Функции управления процессами
- Работа с удаленной памятью
- Разделяемая общая память
- active messages
- threads

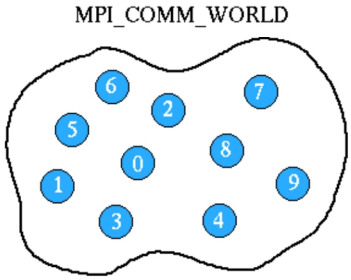
Основные понятия

- MPI-программа – программа запускаемая одновременно на нескольких процессорах
- Каждая копия программы выполняется как отдельный процесс

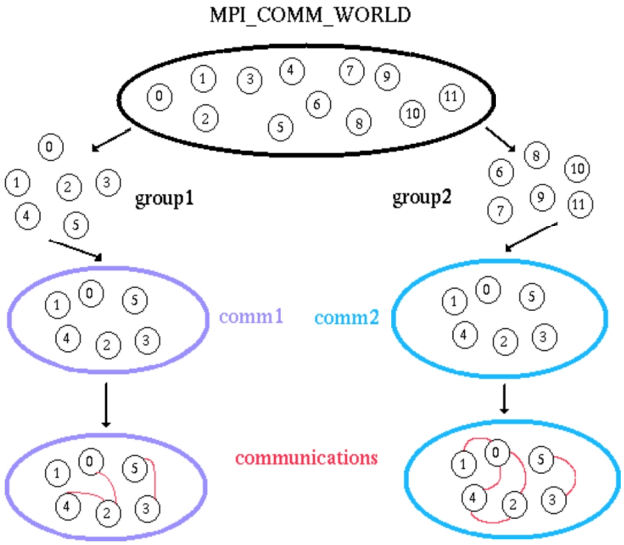


Основные понятия

- Код программы параметризуется номером процесса.
- С помощью служебных функций MPI процесс может получить информацию о количестве одновременно запущенных процессов и свой номер.

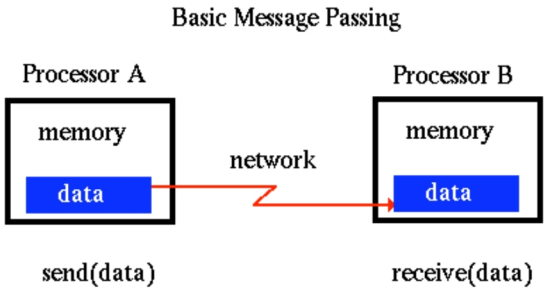


Основные понятия



Основные понятия

- Коммуникационные функции MPI предоставляют процессам MPI-программы различные способы взаимодействия: индивидуальные, групповые.



Основные соглашения

- Регистр символов при указании имен функций (процедур) и именованных констант существенен в Си, и не играет роли в Фортране.
- Все идентификаторы начинаются с префикса **MPI_.** Не рекомендуется использовать собственные идентификаторы, начинающиеся с этого префикса, а также с префиксов **MPID_, MPIR_ и PMPI_,** которые применяются в служебных целях.
- Имена констант (и неизменяемых пользователем переменных) записываются полностью заглавными буквами: **MPI_COMM_WORLD, MPI_FLOAT**.
- В именах функций первая за префиксом буква – заглавная, остальные маленькие: **MPI_Send, MPI_Comm_size.**
- Определение всех именованных констант, прототипов функций и определение типов выполняется в языке Си подключением файла **mpi.h**, а в Фортране – **mpif.h**.

Соответствие между MPI-типами и типами языка Си

тип MPI	тип языка Си
`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`
`MPI_BYTE, MPI_PACKED`	

Соответствие между MPI-типами и типами языка Фортран

тип MPI	Тип языка Фортран
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Первая программа

```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv )
int  argc;
char **argv;
{    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
     printf( "Hello from process %d of %d\n",
       rank, size );
    MPI_Finalize();
    return 0;
}
```

- `int MPI_Init(int *argc, char ***argv)`
- `int MPI_Finalize(void)`
- `int MPI_Comm_size(MPI_Comm comm, int *size)`
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
- `MPI_Get_processor_name (*name,*resultlength)`

Как откомпилировать и запустить?

```
% mpicc -o helloworld helloworld.c
% mpirun -np 4 helloworld
Hello world from process 0 of 4
Hello world from process 3 of 4
Hello world from process 1 of 4
Hello world from process 2 of 4
%
```

На чем запускать?

```
% mpirun -np 8 -machinefile pl helloworld

Содержимое файла pl:
vs21
vs25
vs24
vs23
```

Основная схема

```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv )
int  argc; char **argv;
{    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    if(rank == 0) {/* подготовительные */}
       else {                        /* действия
*/}
    /* собственно реализация алгоритма */
     ...
    MPI_Finalize();
    return 0;
```

MPI_Send

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

MPI_Recv

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

- MPI_ANY_SOURCE
- MPI_ANY_TAG

Основная схема

```
dest = rank + 1;
source = rank - 1;

if(rank == 0) {      ...     }
if(rank == size - 1) {    ...   }

/* код реализующий один шаг алгоритма */



rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
                MPI_COMM_WORLD);
rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,
                MPI_COMM_WORLD, &Stat);
```

Программа пересылки по цепочке



```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv )
int argc;
char **argv;
{
    int rank, value, size;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );

    do { ... } while (value >= 0);

    MPI_Finalize( );
    return 0;
}
```

```
do {
    if (rank == 0) {
        scanf( "%d", &value );
        MPI_Send(&value, 1, MPI_INT,
                 rank + 1, 0,
                 MPI_COMM_WORLD);
    }
    else { MPI_Recv(&value, 1, MPI_INT,
                    rank - 1, 0,
                    MPI_COMM_WORLD, &status);
           if (rank < size - 1)
               MPI_Send(&value, 1, MPI_INT,
                        rank + 1, 0,
                        MPI_COMM_WORLD);
    }
    printf("Process %d got %d\n", rank, value );
   } while (value >= 0);
```

```
% mpicc -o ring ring.c
% mpirun -np 4 ring
10
Process 0 got 10
-1
Process 0 got -1
Process 3 got 10
Process 3 got -1
Process 2 got 10
Process 2 got -1
Process 1 got 10
Process 1 got -1
%
```

```
#include "mpi.h«
#include <stdio.h>
int main(argc,argv)
int argc; char *argv[];
  {int numtasks, rank, dest, source, rc,
       tag=1;
  char inmsg, outmsg='x';
MPI_Status Stat;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
if (rank == 0) {  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest,
         tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source,
         tag, MPI_COMM_WORLD, &Stat);
  }
 else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source,
         tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest,
         tag, MPI_COMM_WORLD);
  }
MPI_Finalize();
}
```

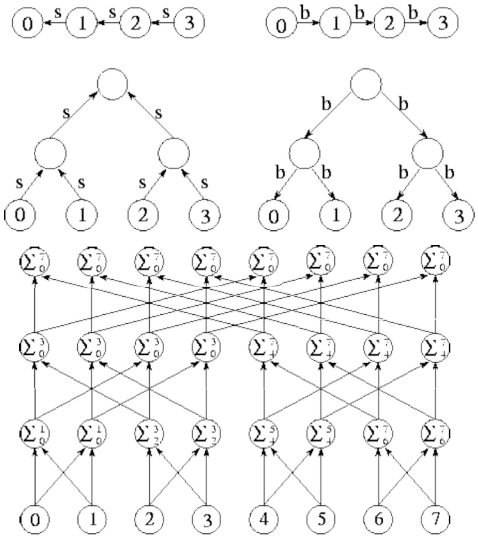
Редукция с последующей общей рассылкой

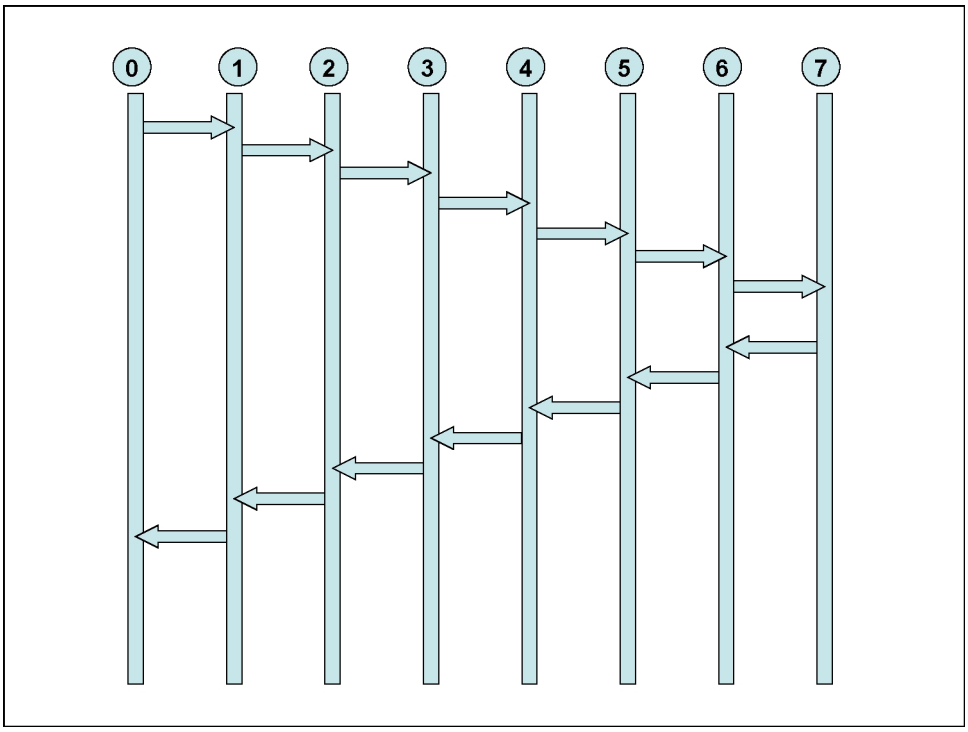
Варианты: 1

2

3

Количество процессов = 8





Этап «Слева направо»

```
Процесс 0:
  sum = data;
  MPI_Send(sum,…rank+1,…);

Процессы 1,…,6:
  MPI_Recv(sum,…rank-1,…);
  sum += data;
  MPI_Send(sum,…rank+1,…);

Процесс 7:
  MPI_Recv(sum,…rank-1,…);
  sum += data;
```

Этап «Справа налево»

```
Процесс 7:
  MPI_Send(sum,…rank-1,…);

Процессы 1,…,6:
  MPI_Recv(sum,…rank+1,…);
  MPI_Send(sum,…rank-1,…);

Процесс 0:
  MPI_Recv(sum,…rank+1,…);
```

Сливаем вместе…

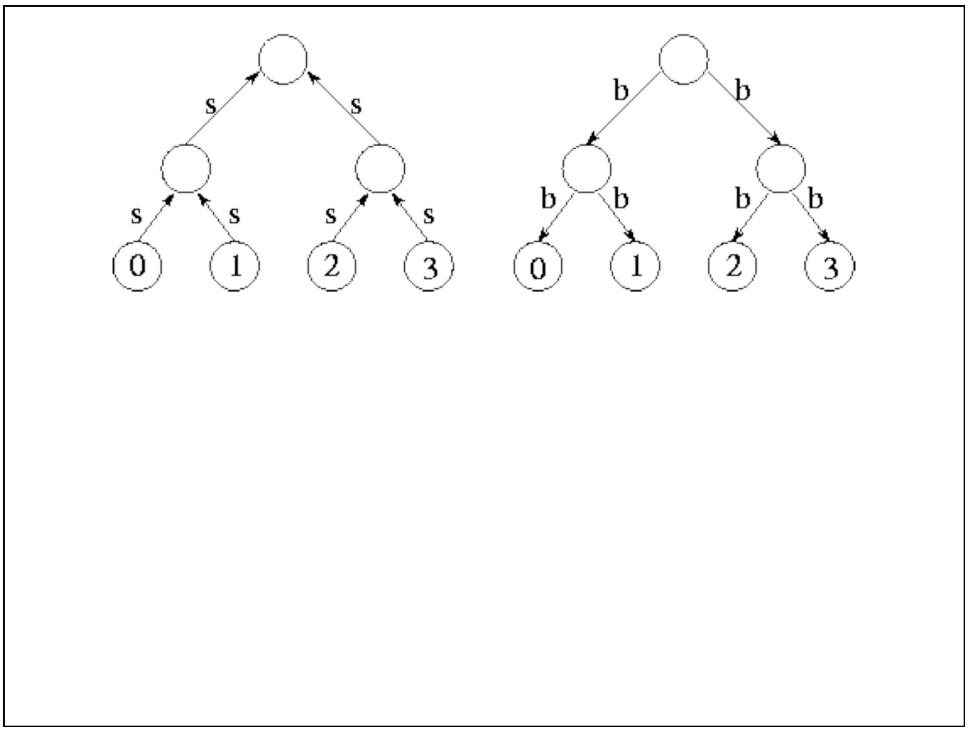
```
Процесс 0:
  sum = data;
  MPI_Send(sum,…rank+1,…);   }
  MPI_Recv(sum,…rank+1,…);   }
Процессы 1,…,6:
  MPI_Recv(sum,…rank-1,…);
  sum += data;
  MPI_Send(sum,…rank+1,…);   }
  MPI_Recv(sum,…rank+1,…);   }
  MPI_Send(sum,…rank-1,…);
Процесс 7:
  MPI_Recv(sum,…rank-1,…);
  sum += data;
  MPI_Send(sum,…rank-1,…);
```

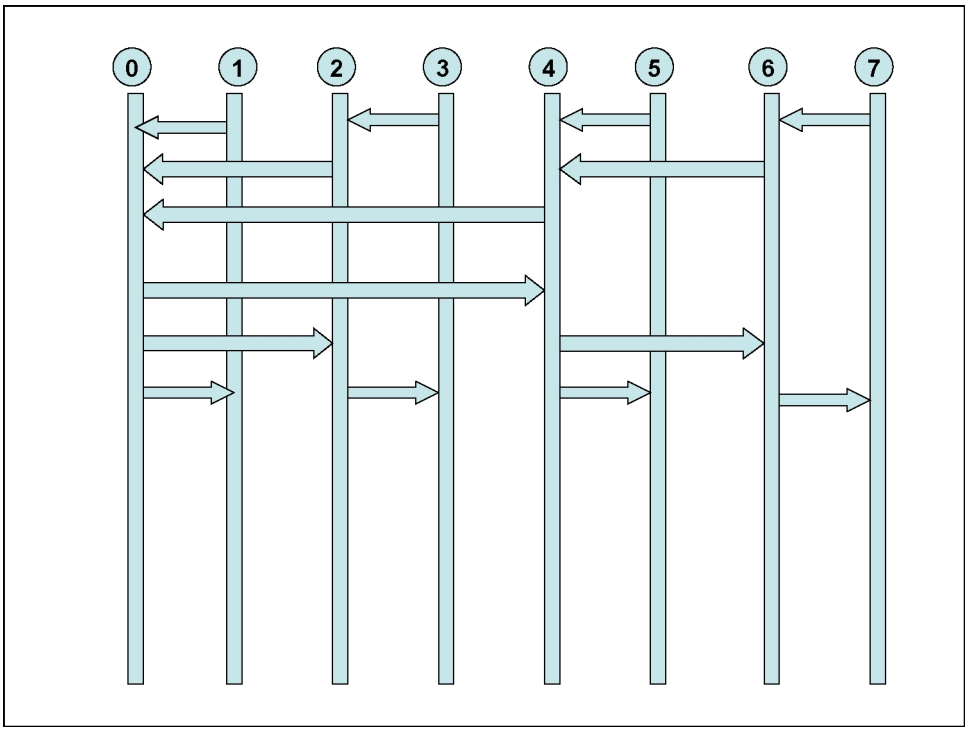
MPI_Sendrecv

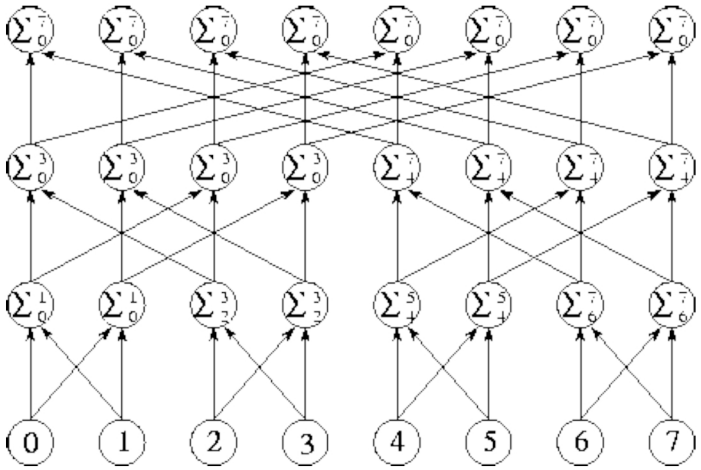
MPI_Sendrecv(*sendbuf,sendcount,sendtype,
dest,sendtag,
***recvbuf,recvcount,recvtype,**
source,recvtag,comm,*status)

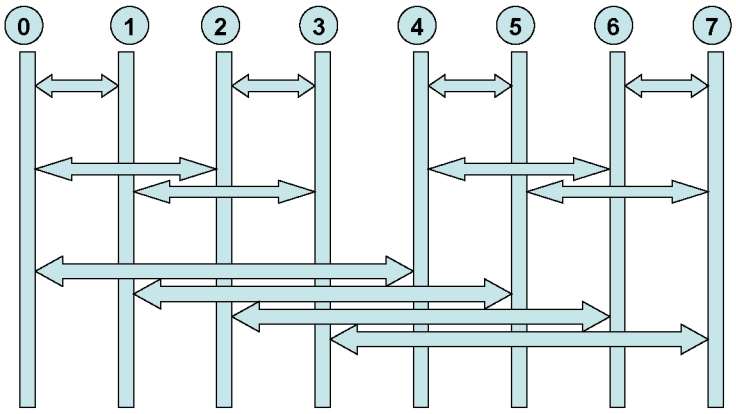
MPI_SENDRECV(sendbuf,sendcount,sendtype,
dest,sendtag,
recvbuf,recvcount,recvtype,
source,recvtag,comm,status,ierr)

```
if(rank == 0) {
  sum = data;
  MPI_Send(sum,…rank+1,…);
  MPI_Recv(sum,…rank+1,…);
} else {
  MPI_Recv(sum,…rank-1,…);
  sum += data;
  if(rank != n-1) {
     MPI_Send(sum,…rank+1,…);
     MPI_Recv(sum,…rank+1,…);
  }
  MPI_Send(sum,…rank-1,…);
}
```


s
s
s
s
s
s
0
1
2
3
b
b
b
b
b
b
0
1
2
3


0
1
2
3
4
5
6
7


0
1
2
3
4
5
6
7


0
1
2
3
4
5
6
7

```
int dest[][]={{1,0,3,2,5,4,7,6},
              {2,3,0,1,6,7,4,5},
              {4,5,6,7,0,1,2,3}};

sum = data;
for(int i=0; i < 3; i++) {
  MPI_Send(sum,…dest[i][rank],…);
  MPI_Recv(r_data,…dest[i][rank],…);
  sum += r_data;
}
```

Подробнее об операциях пересылки сообщений

Blocking sends	`MPI_Send(buffer,count,type,dest,tag, comm)`
Non-blocking sends	`MPI_Isend(buffer,count,type,dest,tag, comm,request)`
Blocking receive	`MPI_Recv(buffer,count,type,source,tag, comm,status)`
Non-blocking receive	`MPI_Irecv(buffer,count,type,source,tag, comm,request)`

MPI_Ssend

```
ierr = MPI_Ssend (*buf,count,
                  datatype,
                  dest,tag,comm)

MPI_SSEND (buf,count,datatype,
           dest,tag,comm,ierr)
```

MPI_Bsend

```
MPI_Bsend (*buf,count,datatype,
           dest,tag,comm)
MPI_BSEND (buf,count,datatype,
           dest,tag,comm,ierr)
```

MPI_Buffer_attach / MPI_Buffer_detach

```
MPI_Buffer_attach (*buffer,size)
MPI_Buffer_detach (*buffer,size)
MPI_BUFFER_ATTACH (buffer,size,ierr)
MPI_BUFFER_DETACH (buffer,size,ierr)
```

MPI_Rsend

```
MPI_Rsend(*buf,count,datatype,
          dest,tag,comm)
MPI_RSEND(buf,count,datatype,
          dest,tag,comm,ierr)
```

MPI_Probe

```
MPI_Probe (source,tag,comm,*status)
```

MPI_Sendrecv

```
MPI_Sendrecv(*sendbuf,sendcount,sendtype,
             dest,sendtag,
             *recvbuf,recvcount,recvtype,
             source,recvtag,comm,*status)

MPI_SENDRECV(sendbuf,sendcount,sendtype,
             dest,sendtag,
             recvbuf,recvcount,recvtype,
             source,recvtag,comm,status,ierr)
```

MPI_Isend

```
MPI_Isend(*buf,count,datatype,
          dest,tag,comm,*request)
          MPI_ISEND (buf,count,datatype,
          dest,tag,comm,request,ierr)
```

MPI_Irecv

```
MPI_Irecv(*buf,count,datatype,
          source,tag,comm,*request)
          MPI_IRECV (buf,count,datatype,
          source,tag,comm,request,ierr)
```

MPI_Issend

```
MPI_Issend(*buf,count,datatype,
           dest,tag,comm,*request)
           MPI_ISSEND (buf,count,datatype,
           dest,tag,comm,request,ierr)
```

MPI_Ibsend

```
MPI_Ibsend(*buf,count,datatype,
           dest,tag,comm,*request)
           MPI_IBSEND (buf,count,datatype,
           dest,tag,comm,request,ierr)
```

MPI_Irsend

```
MPI_Irsend(*buf,count,datatype,
           dest,tag,comm,*request)
           MPI_IRSEND (buf,count,datatype,
           dest,tag,comm,request,ierr)
```

MPI_Test
MPI_Testany
MPI_Testall
MPI_Testsome

```
MPI_Test(*request,*flag,*status)

MPI_Testany (count,*array_of_requests,
          *index,*flag,*status)

MPI_Testall (count,*array_of_requests,
          *flag,*array_of_statuses)

MPI_Testsome (incount,*array_of_requests,
          *outcount,*array_of_offsets,
          *array_of_statuses)
```

MPI_Wait
MPI_Waitany
MPI_Waitall
MPI_Waitsome

```
MPI_Wait (*request,*status)
MPI_Waitany (count,*array_of_requests,*index,
             *status)
 MPI_Waitall (count,*array_of_requests,
             *array_of_statuses)
 MPI_Waitsome (incount,*array_of_requests,
             *outcount,
             *array_of_offsets,
             *array_of_statuses)
```

Библиотека MPI

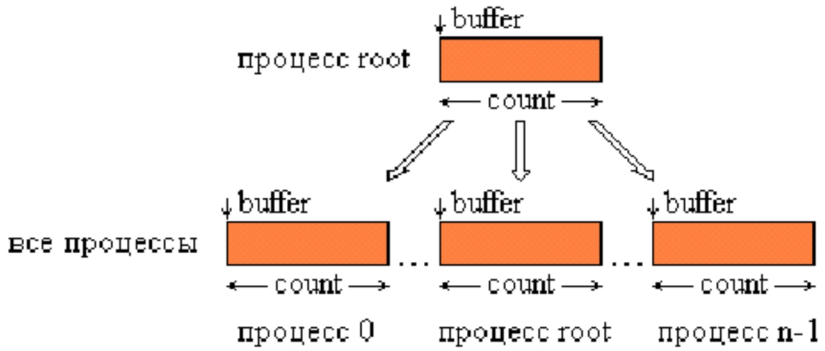
Коллективные операции

Функция синхронизации процессов

int MPI_Barrier(MPI_Comm comm)

Широковещательная рассылка данных

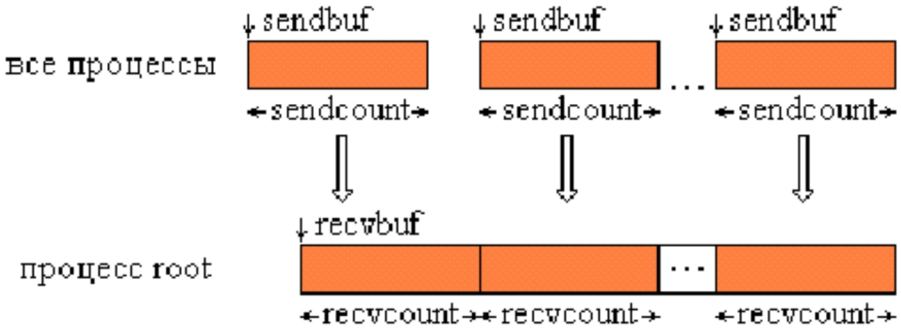
int MPI_Bcast(void* buffer, int count,
MPI_Datatype datatype,
int root, MPI_Comm comm)



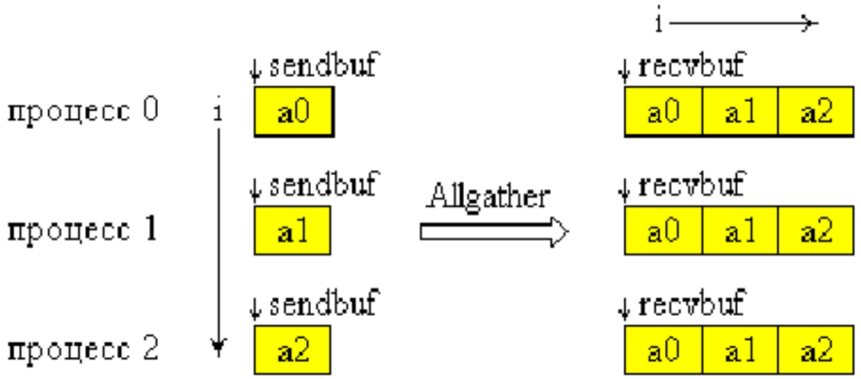
Функции сбора блоков данных от всех процессов группы

- MPI_Gather
- MPI_Allgather
- MPI_Gatherv
- MPI_Allgatherv.

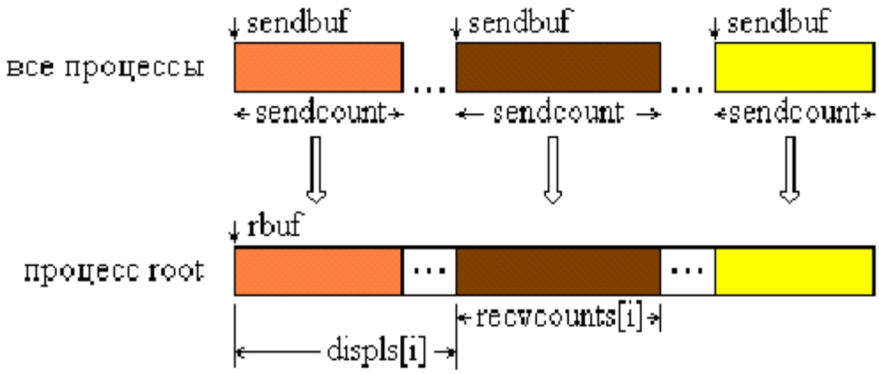
int MPI_Gather(void* sendbuf,
int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount,
MPI_Datatype recvtype, int root,
MPI_Comm comm)



int MPI_Allgather(void* sendbuf, int
sendcount, MPI_Datatype sendtype, void*
recvbuf, int recvcount, MPI_Datatype
recvtype, MPI_Comm comm)



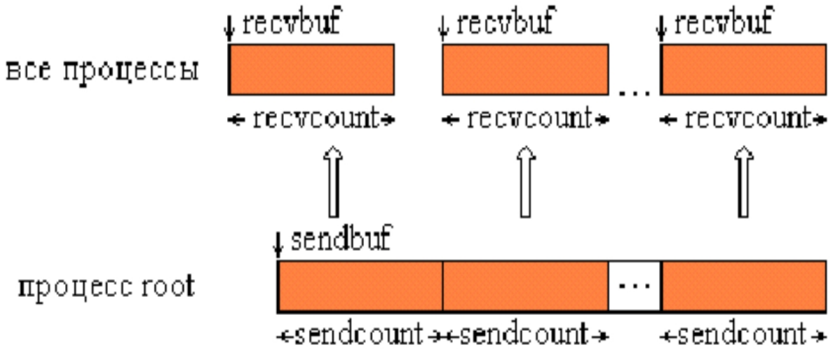
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* rbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)



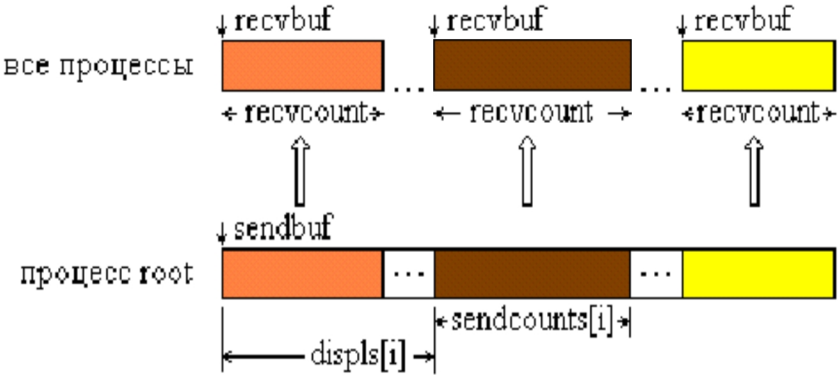
Функции распределения блоков данных по всем процессам группы

- MPI_Scatter
- MPI_Scatterv

int MPI_Scatter(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf,
int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm)



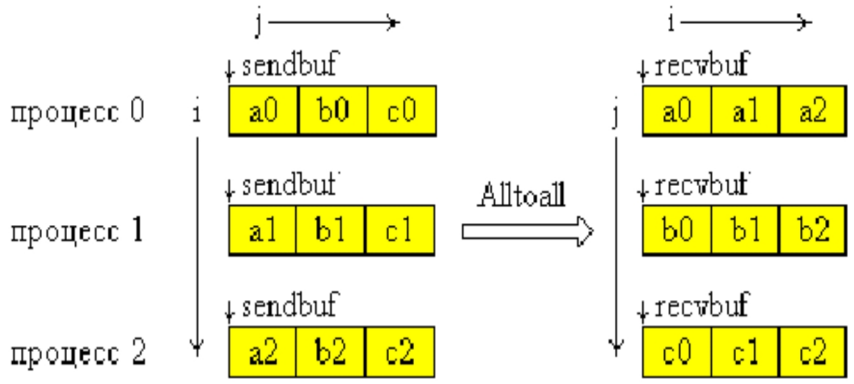
int MPI_Scatterv(void* sendbuf, int *sendcounts,
int *displs, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm comm)



Совмещенные коллективные операции

- MPI_Alltoall
- MPI_Alltoallv

int MPI_Alltoall(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf,
int recvcount, MPI_Datatype recvtype,
MPI_Comm comm)

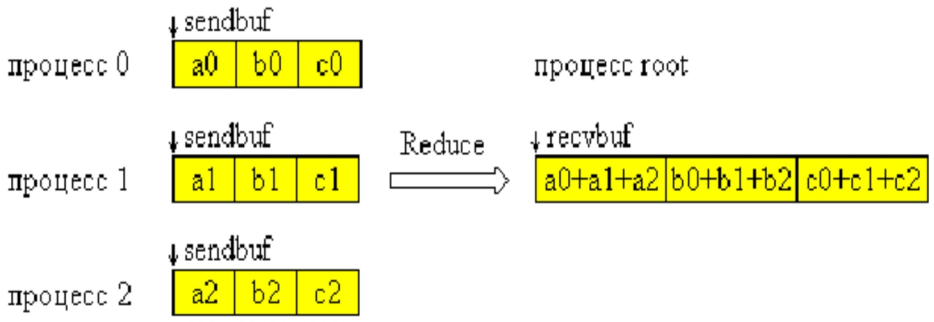


Глобальные вычислительные операции над распределенными данными

- глобальные операции редукции

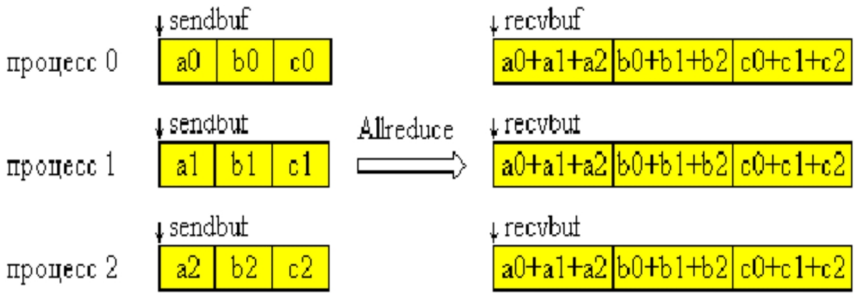
- MPI_Reduce
- MPI_Allreduce
- MPI_Scan
- MPI_Reduce_scatter

int MPI_Reduce(void* sendbuf, void* recvbuf,
int count, MPI_Datatype datatype,
MPI_Op op, int root, MPI_Comm comm)

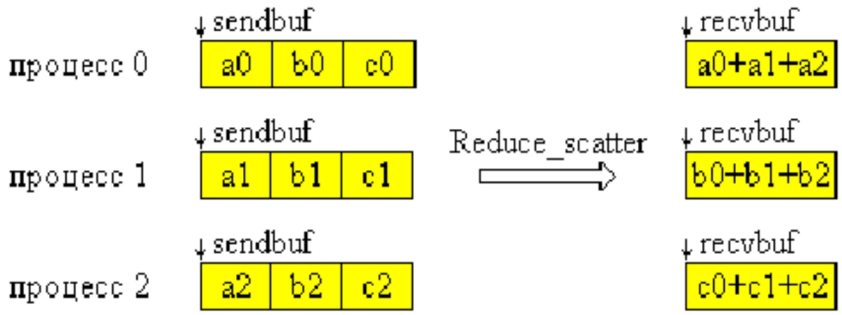


Название	Операция	Разрешенные типы
MPI_MAX MPI_MIN	Максимум Минимум	C integer, FORTRAN integer, Floating point
MPI_SUM MPI_PROD	Сумма Произведение	C integer, FORTRAN integer, Floating point, Complex
MPI_LAND MPI_LOR MPI_LXOR	Логическое AND Логическое OR Логическое исключающее OR	C integer, Logical
MPI_BAND MPI_BOR MPI_BXOR	Поразрядное AND Поразрядное OR Поразрядное исключающее OR	C integer, FORTRAN integer, Byte
MPI_MAXLOC MPI_MINLOC	Максимальное значение и его индекс Минимальное значение и его индекс	Специальные типы для этих функций

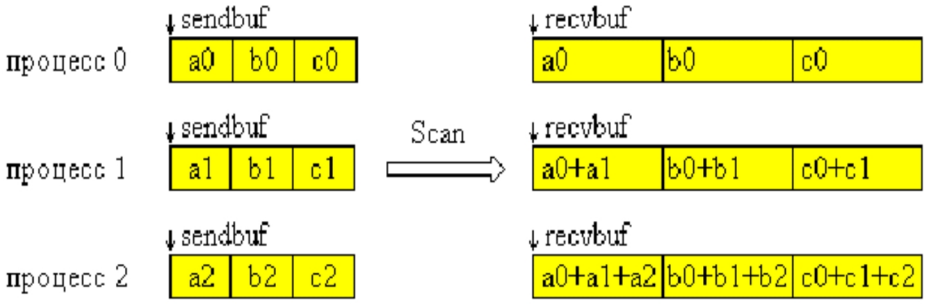
```
int MPI_Allreduce(void* sendbuf,
    void* recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op,
    MPI_Comm comm)
```



int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)



int MPI_Scan(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)



Библиотека MPI

ПРОИЗВОДНЫЕ ТИПЫ ДАННЫХ И ПЕРЕДАЧА УПАКОВАННЫХ ДАННЫХ

Механизмы эффективной пересылки данные разных типов

- создание производных типов для использования в коммуникационных операциях вместо предопределенных типов MPI
- пересылка упакованных данных (процесс-отправитель упаковывает пересылаемые данные перед их отправкой, а процесс-получатель распаковывает их после получения)

Производные типы данных

- Производный тип строится из предопределенных типов MPI и ранее определенных производных типов с помощью специальных функций-конструкторов MPI_Type_contiguous, MPI_Type_vector, MPI_Type_hvector, MPI_Type_indexed, MPI_Type_hindexed, MPI_Type_struct

Производные типы данных

- регистрируются вызовом функции MPI_Type_commit
- ненужные типы уничтожаются функцией MPI_Type_free
- Предопределенные типы MPI считаются зарегистрированными

Характеристики типов

- Протяженность типа – сколько байт переменная данного типа занимает в памяти
  (адрес последней ячейки данных - адрес первой ячейки данных + длина последней ячейки данных)

- MPI_Type_extent

Характеристики типов

- Размер типа определяет количество реально передаваемых байт в коммуникационных операциях
- Эта величина равна сумме длин всех базовых элементов определяемого типа
- опрашивается подпрограммой MPI_Type_size

```
int MPI_Type_extent(MPI_Datatype datatype,
                    MPI_Aint *extent)

int MPI_Type_size(MPI_Datatype datatype,
                  int *size)
```

```
int MPI_Type_contiguous(int count,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype)
```

oldtype = MPI_REAL

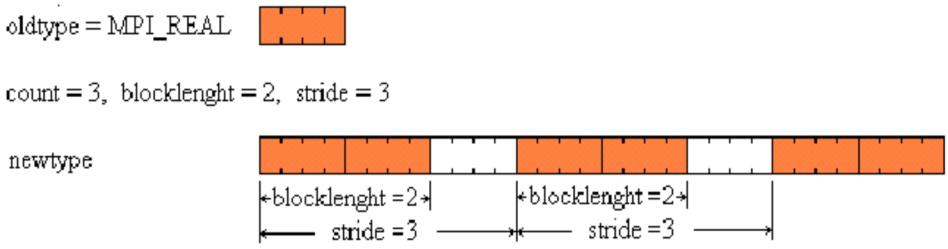


count = 4

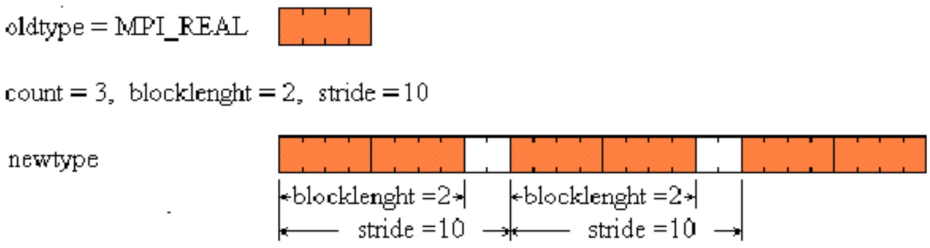
newtype



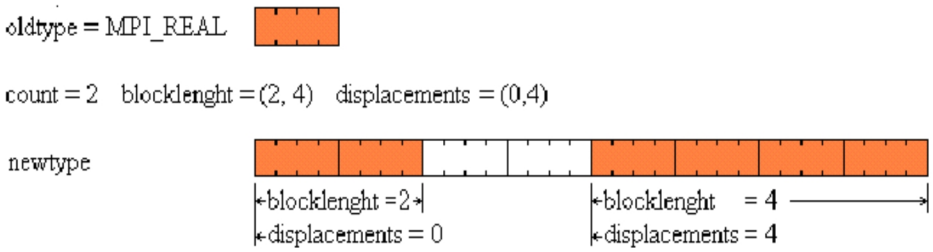
int MPI_Type_vector(int count, int blocklength,
int stride, MPI_Datatype oldtype,
MPI_Datatype *newtype)



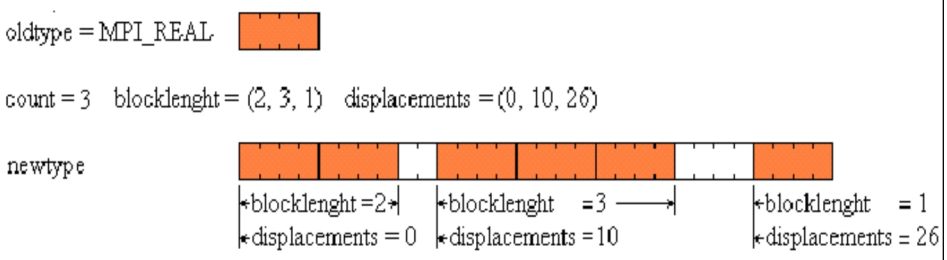
int MPI_Type_hvector(int count,
int blocklength, MPI_Aint stride,
MPI_Datatype oldtype,
MPI_Datatype *newtype)



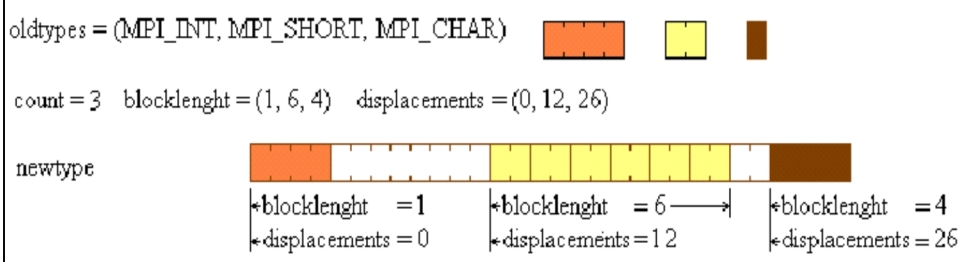
```
int MPI_Type_indexed(int count,
        int *array_of_blocklengths,
        int *array_of_displacements,
        MPI_Datatype oldtype,
        MPI_Datatype *newtype)
```




```
int MPI_Type_hindexed(int count,
        int *array_of_blocklengths,
        MPI_Aint *array_of_displacements,
        MPI_Datatype oldtype,
        MPI_Datatype *newtype)
```



```
int MPI_Type_struct(int count,
            int *array_of_blocklengths,
            MPI_Aint *array_of_displacements,
            MPI_Datatype *array_of_types,
            MPI_Datatype *newtype)
```




```
int MPI_Type_commit(MPI_Datatype
  *datatype)

int MPI_Type_free(MPI_Datatype *datatype)
```

- Функция MPI_Type_free устанавливает описатель типа в состояние MPI_DATATYPE_NULL

```
#include <mpi.h>
#include <stdio.h>
#define  SIZE  7
#define  halfSize  (SIZE+1)/2
int main( int argc, char **argv )
{
    int a[ SIZE ][ SIZE ],
        b[ halfSize ][ halfSize ],
        c[ SIZE ][ SIZE ];
    int rank, i, j;
    MPI_Aint typeExtent;
    MPI_Datatype
        oneSlice, twoSlice,
        oneRow, rowByRow;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
```

```
    if( rank != 0 )
        goto done;
    for( i=0; i<SIZE; i++ )
        for( j=0; j<SIZE; j++ )
            a[i][j] = 10 * (i+1) + (j+1);
    puts("Sended:");
    for( i=0; i<SIZE; i++ ) {
        for( j=0; j<SIZE; j++ )
            printf("%4d", a[i][j] );
        puts("");
    }
    puts("");
```

```
/*-------------------------------------------*
 *Первый пример: выделение и пересылка решетки *
 *-------------------------------------------*/
     /* Создаем тип из элементов вида 'x.'
      * #1. Количество элементов в новом типе
               = половине числа столбцов
      * #2. Каждый элемент содержит
               одно целое число
      * #3. Начала элементов отстоят друг от
               друга на две ячейки типа int
      * => Новый тип накрывает РОВНО ОДНУ строку
               матрицы шаблоном 'x.x.x.x.'
      */
    MPI_Type_vector( halfSize, 1, 2,
                     MPI_INT, &oneSlice );
```

```
/* Создаем тип из элементов вида 'x.x.x.x.x.x.x.x.'
   #1. Количество элементов в новом типе =
          половине числа строк
   #2. Каждый элемент содержит одну ячейку
          типа 'oneSlice'
   #3. Начала элементов отстоят друг от друга
          на две строки
   => Новый тип накрывает матрицу решеткой
          с нечетными координатами
 */

  MPI_Type_hvector( halfSize,
                    1,
                    SIZE*2*sizeof(int),
                    oneSlice,
                    &twoSlice );
```

```
/* Регистрируем тип 'twoSlice'
 * Поскольку тип 'oneSlice' используется
 * только для порождения других типов,
 * его регистрировать не надо.
 */

    MPI_Type_commit( &twoSlice );

/* На этом подготовительная работа,
 * выполняемая РОВНО ОДИН РАЗ, завершена.
 */
```

```
/* Посылаем сами себе.
 * Отправляем одну переменную типа 'twoSlice',
 * расположенную по адресу матрицы 'a'.
 * Принимаем поток целых чисел и размещаем его
 * по адресу матрицы 'b'
 */
    MPI_Sendrecv( a, 1, twoSlice, rank, 0,
                  b, halfSize*halfSize,
                  MPI_INT, rank, 0,
                  MPI_COMM_WORLD, &status );
```

```
/* Печать принятой подматрицы
 */
    puts("Received sub-matrix:");
    for( i=0; i<halfSize; i++ ) {
        for( j=0; j<halfSize; j++ )
            printf("%4d", b[i][j] );
        puts("");
    }
    puts("");

    MPI_Type_extent( oneSlice, &typeExtent );
    MPI_Type_size  ( oneSlice, &typeSize );
    MPI_Type_extent( twoSlice, &typeExtent );
    MPI_Type_size  ( twoSlice, &typeSize );
```

```
/* Очистка описателей. MPI удалит соотвествующие
 * данные из внутренних таблиц и
 * сбросит описатели в MPI_TYPE_NULL
 */
    MPI_Type_free( &oneSlice );
    MPI_Type_free( &twoSlice );
```

```
/*--------------------------------------------*
 *Второй пример: пересылка с транспонированием *
 *--------------------------------------------*
 * В оперативной памяти матрица хранится
 * построчно: строка1,строка2,...
 * Создаем тип-шаблон для быстрого выделения
 * из матрицы одного столбца:
 * #1. Число элементов = количеству строк,
 *        т.е. высоте столбца
 * #2. В каждой строке нас интересует
 *        одна ячейка
 * #3. Элементы расположены в памяти с шагом,
 *        равным длине строки
 */

    MPI_Type_vector( SIZE, 1, SIZE,
                     MPI_INT, &oneRow );
```

```
/* Получаем тип для постолбцового представления
 * матрицы:
 * #1. Число элементов = количеству столбцов
 * #2. Каждый элемент содержит один столбец,
 *        т.е. ячейку типа 'oneRow'
 * #3. Начала элементов (т.е. столбцов) отстоят
 *     друг от друга на расстоянии в одну ячейку
 *     типа int.
 */
    MPI_Type_hvector( SIZE, 1, sizeof(int),
                      oneRow, &rowByRow );

    MPI_Type_commit( &rowByRow );
/* Тип зарегистрирован,
 * подготовительная работа завершена.
 */
```

```
/* Передаем с одновременным транспонированием.
 * Передается переменная типа 'rowByRow',
 * находящаяся по адресу матрицы 'a'.
 * Данные уходят столбец за столбцом.
 * Принимается поток целых чисел,
 * размещаемый по адресу матрицы 'C'.
 * Данные размещаются непрерывно,
 * то есть построчно!
 */

    MPI_Sendrecv( a, 1, rowByRow, rank, 0,
                  c, SIZE*SIZE, MPI_INT, rank,
                  0, MPI_COMM_WORLD, &status );
```

```
   /* Печать принятой матрицы
    */

       puts("Received transposed matrix:");
       for( i=0; i<SIZE; i++ ) {
           for( j=0; j<SIZE; j++ )
               printf("%4d", c[i][j] );
           puts("");
       }
       puts("");

   MPI_Type_extent( oneRow, &typeExtent );
   MPI_Type_size  ( oneRow, &typeSize ); */
   MPI_Type_extent( rowByRow, &typeExtent );
   MPI_Type_size  ( rowByRow, &typeSize ); */
```

```
    /* Очистка описателей.
     */

        MPI_Type_free( &oneRow );
        MPI_Type_free( &rowByRow );

    /*----------------------*
     *  Завершение работы   *
     *----------------------*/

    done:
        MPI_Finalize();
        return 0;
    }
```

Результат запуска

```
spop@h1:~$ mpicc -o m matrix.c
spop@h1:~$ mpirun -np 7 m
Sended:
  11  12  13  14  15  16  17
  21  22  23  24  25  26  27
  31  32  33  34  35  36  37
  41  42  43  44  45  46  47
  51  52  53  54  55  56  57
  61  62  63  64  65  66  67
  71  72  73  74  75  76  77
```

```
Received sub-matrix:
  11  13  15  17
  31  33  35  37
  51  53  55  57
  71  73  75  77
'oneSlice' has extent =  28
'twoSlice' has extent = 196
```

```
Received transposed matrix:
  11  21  31  41  51  61  71
  12  22  32  42  52  62  72
  13  23  33  43  53  63  73
  14  24  34  44  54  64  74
  15  25  35  45  55  65  75
  16  26  36  46  56  66  76
  17  27  37  47  57  67  77
'oneRow'   has extent = 172
'rowByRow' has extent = 196
spop@h1:~$
```

Передача упакованных данных

```
int MPI_Pack(void* inbuf, int incount,
    MPI_Datatype datatype, void *outbuf,
    int outsize, int *position, MPI_Comm comm)
int MPI_Pack_size(int incount, MPI_Datatype
    datatype, MPI_Comm comm, int *size)

int MPI_Unpack(void* inbuf, int insize,
    int *position, void *outbuf, int outcount,
    MPI_Datatype datatype, MPI_Comm comm)
```

/* Упаковка данных */

```
if (myrank == 0) {
    position = 0;
    MPI_Pack(&x, 1, MPI_DOUBLE, buff,
    100, &position, MPI_COMM_WORLD);
    MPI_Pack(&y, 1, MPI_DOUBLE, buff,
    100, &position, MPI_COMM_WORLD);
    MPI_Pack(a, 2, MPI_INT, buff, 100,
    &position, MPI_COMM_WORLD);
}
```

/* Рассылка упакованного сообщения */

```
MPI_Bcast(buff, position, MPI_PACKED, 0,
    MPI_COMM_WORLD);
```

/* Распаковка сообщения во всех процессах */

```
if (myrank != 0) {
  position = 0;
  MPI_Unpack(buff, 100, &position, &x, 1,
  MPI_DOUBLE, MPI_COMM_WORLD);
  MPI_Unpack(buff, 100, &position, &y, 1,
  MPI_DOUBLE, MPI_COMM_WORLD);
  MPI_Unpack(buff, 100, &position, a, 2,
  MPI_INT, MPI_COMM_WORLD);
}
```

Библиотека MPI

Работа с группами и коммуникаторами

Основные понятия

Два взаимосвязанных механизма:

- функции для работы с группами процессов как упорядоченными множествами
- функции для работы с коммуникаторами, для создания новых коммуникаторов как описателей новых областей связи

Основные понятия (группа)

- *Группа* – упорядоченное множество процессов
- Каждый процесс идентифицируется переменной целого типа (непрерывный ряд, начинающийся с 0)
- Специальный тип данных MPI_Group
- Набор функций для работы с переменными и константами типа MPI_Group

Основные понятия (группа)

- Две предопределенные группы:
  - MPI_GROUP_EMPTY
  - MPI_GROUP_NULL
- Нет группы, соответствующей коммуникатору MPI_COMM_WORLD
- Созданная группа не может быть модифицирована (расширена или усечена)
- Можно только создать новую группу

Основные понятия (коммуникатор)

- **Коммуникатор** – скрытый объект с некоторым набором атрибутов, правилами его создания, использования и уничтожения
- Коммуникатор описывает некоторую область связи
- Два предопределенных коммуникатора:
  - **MPI_COMM_WORLD**
  - **MPI_COMM_SELF**

Основные понятия (коммуникатор)

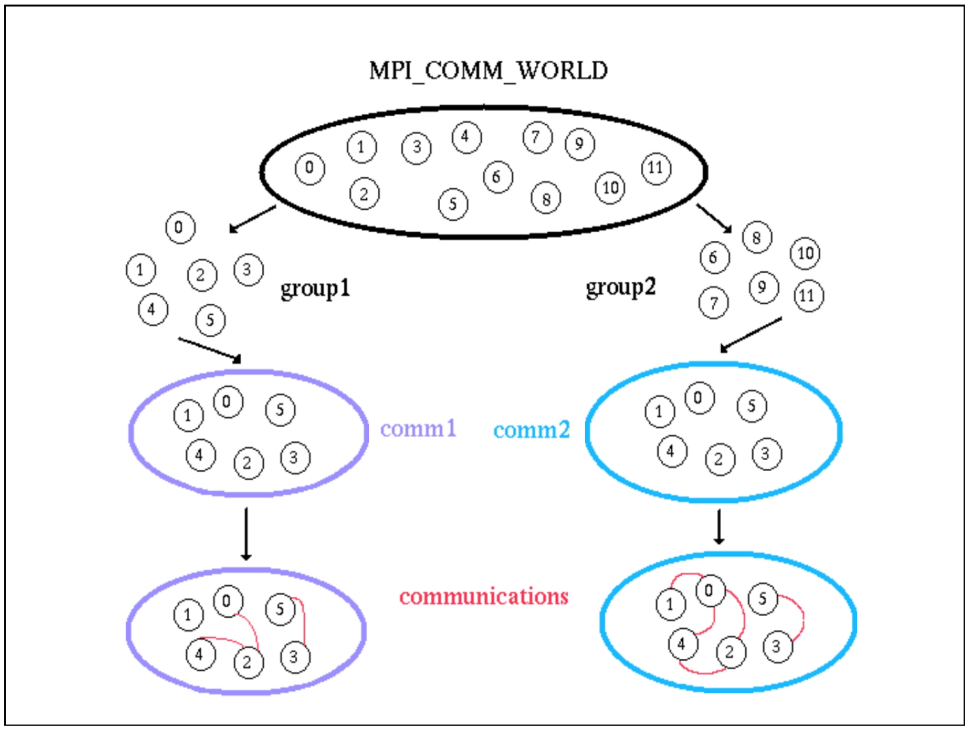
- Одной и той же области связи может соответствовать несколько коммуникаторов, но они не являются тождественными и не могут участвовать во взаимном обмене сообщениями.
- Если данные посылаются через один коммуникатор, процесс-получатель может получить их только через тот же самый коммуникатор.

Типичное использование

1. **MPI_Comm_group**: Получаем описатель глобальной группы, содержащей все процессы из MPI_COMM_WORLD
2. **MPI_Group_incl**: Формируем новую группу как подмножество глобальной группы
3. **MPI_Comm_create**: Создаем новый коммуникатор для новой группы

Типичное использование

4. **MPI_Comm_rank**: Получаем новый номер *rank* процесса в новом коммуникаторе
5. **В**ыполняем передачу данных
6. **MPI_Comm_free и MPI_Group_free**: Освобождаем описатели нового коммуникатора и новой группы



Определение числа процессов в группе

```
MPI_Group_size(MPI_Group group,
               int *size)
```

Определение номера процесса в группе

```
MPI_Group_rank(MPI_Group group,
               int *rank)
```

Установка соответствия между номерами процессов в двух группах

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

ranks1 – массив номеров процессов из 1-й группы

ranks2 – номера тех же процессов во 2-ой группе

Создание групп

```
MPI_Comm_group(MPI_Comm comm,
                        MPI_Group
*group)
```

Создание групп

```
MPI_Group_union(
          MPI_Group group1,
          MPI_Group group2,
          MPI_Group *newgroup)

MPI_Group_intersection(group1,
          group2, &newgroup)
MPI_Group_difference(group1,
          group2, &newgroup)
```

Создание групп

```
MPI_Group_incl(MPI_Group group,
            int n, int *ranks,
            MPI_Group *newgroup)

MPI_Group_excl(MPI_Group group,
            int n, int *ranks,
            MPI_Group *newgroup)
```

Создание групп

```
MPI_Group_range_incl(
       MPI_Group group, int n,
       int ranges[][3],
       MPI_Group *newgroup)
MPI_Group_range_excl(
       MPI_Group group, int n,
       int ranges[][3],
       MPI_Group *newgroup)
```

Триплет имеет вид:

нижняя граница, верхняя граница, шаг.

Уничтожение созданных групп

```
MPI_Group_free(MPI_Group *group)
```

Функции доступа к коммуникатору

```
int MPI_Comm_size(
        MPI_Comm comm, int *size)

int MPI_Comm_rank(
        MPI_Comm comm, int *rank)
```

Сравнение двух коммуникаторов

```
MPI_Comm_compare( MPI_Comm comm1,
     MPI_Comm comm2, int *result)
```

Возможные значения результата сравнения:

- MPI_IDENT (один и тот же объект)
- MPI_CONGRUENT (две области связи, одни и те же атрибуты группы)
- MPI_SIMILAR (другое упорядочивание групп)
- MPI_UNEQUAL

Дублирование коммуникатора

```
MPI_Comm_dup(MPI_Comm comm,
             MPI_Comm *newcomm)
```

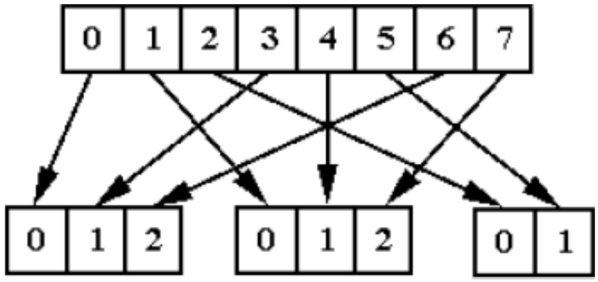
Создание коммуникатора

```
MPI_Comm_create(MPI_Comm comm,
                MPI_Group group,
                MPI_Comm *newcomm)
```

Расщепление коммуникатора

```
MPI_Comm_split(MPI_Comm comm,
               int color,
               int key,
               MPI_Comm *newcomm)
```

```
MPI_comm comm, newcomm;
int myid, color;
……
MPI_Comm_rank(comm, &myid);
color = myid%3;
MPI_Comm_split(comm, color, myid,
                &newcomm);
```



Уничтожение коммуникатора

```
MPI_Comm_free(MPI_Comm *comm)
```

Библиотека MPI

Виртуальные топологии процессов

Основные понятия

- Топология – механизм сопоставления процессам коммуникатора альтернативной схемы адресации
- В MPI топологии – виртуальные
- В MPI два типа топологий:
  - Декартова топология
  - Топология графа

Определение типа топологии

```
MPI_Topo_test(MPI_Comm comm,
              int *status)
```

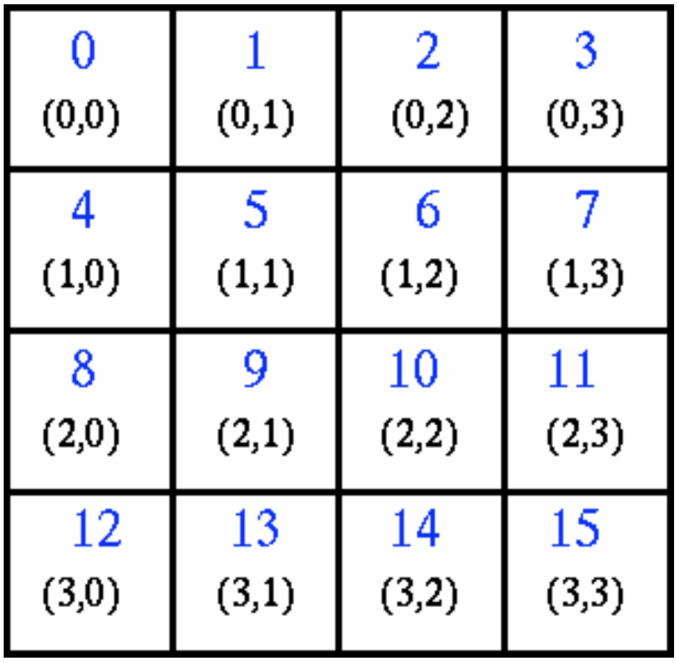
Возможные значения `status`:

`MPI_GRAPH`
`MPI_CART`
`MPI_UNDEFINED`

Создание коммуникатора с декартовой топологией

```
MPI_Cart_create(MPI_Comm comm_old,
    int ndims, int *dims,
    int *periods, int reorder,
    MPI_Comm *comm_cart)
```

Если размеры заказываемой сетки больше имеющегося в группе числа процессов, то функция завершается аварийно



Определение оптимальной конфигурации сетки

```
MPI_Dims_create(int nnodes,
     int ndims, int *dims)
```

Вычисляются только те компоненты для которых `dims[i]=0`

Создается максимально равномерное распределение процессов вдоль направлений, выстраивая их по убыванию: для **12 –> 4x3x1**

Определение оптимальной конфигурации сетки

dims перед вызовом	Параметры функции `MPI_Dims_create`	**dims** после вызова
`( 0, 0, 0 )`	`( 6, 3, dims )`	`( 3, 2, 1 )`
`( 0, 0, 0 )`	`( 7, 3, dims )`	`( 7, 1, 1 )`
`( 0, 3, 0 )`	`( 6, 3, dims )`	`( 2, 3, 1 )`
`( 0, 3, 0 )`	`( 7, 3, dims )`	Ошибка!

Определение оптимальной конфигурации сетки

```
MPI_Dims_create(int nnodes,
     int ndims, int *dims)
```

Вычисляются только те компоненты для которых `dims[i]=0`

Создается максимально равномерное распределение процессов вдоль направлений, выстраивая их по убыванию: для **12 –> 4x3x1**

Получение числа измерений и другой информации

```
MPI_Cartdim_get(MPI_Comm comm,
     int *ndims)

MPI_Cart_get(MPI_Comm comm,
     int ndims, int *dims,
     int *periods, int *coords)
```

Получение идентификатора процесса по его координатам

```
MPI_Cart_rank(MPI_Comm comm,
     int *coords, int *rank)
```

Для измерений с периодическими граничными условиями будет выполняться приведение к основной области определения

`0 <= coords(i) < dims(i)`

Определение координат процесса по его идентификатору

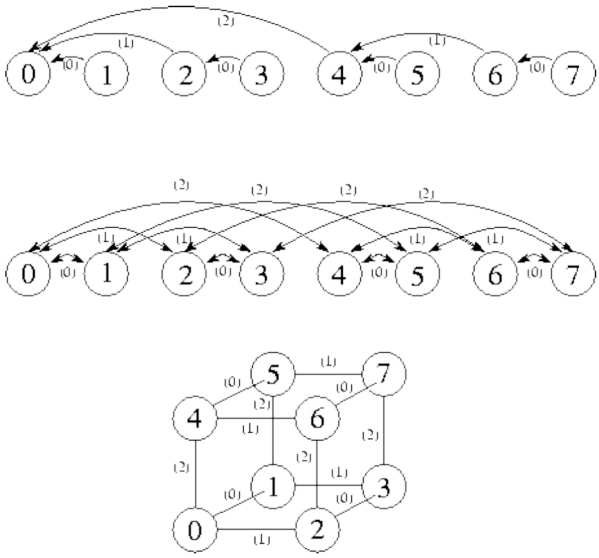
```
MPI_Cart_coords(MPI_Comm comm,
     int rank, int ndims,
     int *coords)
```

Определение параметров сдвига данных

```
MPI_Cart_shift(MPI_Comm comm,
     int direction, int disp,
     int *rank_source,
     int *rank_dest)
```

`MPI_PROC_NULL`

Пример: Allreduce




```
int dest[][]={{1,0,3,2,5,4,7,6},
              {2,3,0,1,6,7,4,5},
              {4,5,6,7,0,1,2,3}};

sum = data;
for(int i=0; i < 3; i++) {
  MPI_Send(sum,…dest[i][rank],…);
  MPI_Recv(r_data,…dest[i][rank],…);
  sum += r_data;
}
```

```
MPI_Comm d_topo;
int dims[3] = {2,2,2};
int periods[3] = {1,1,1};
int reorder = 0, src, dest;
MPI_Cart_create(MPI_COMM_WORLD, 3, dims,
  periods, reorder, &d_topo);

sum = data;
for(int i=0; i < 3; i++) {
  MPI_Cart_shift(d_topo, i, 1, &src, &dest);
  MPI_Send(sum,…dest,…);
  MPI_Recv(r_data,…src,…);
  sum += r_data;
}
```

Выделение подпространства в декартовой топологии

MPI_Cart_sub(MPI_Comm comm,
int *remain_dims,
MPI_Comm *newcomm)

remain_dims[i] = true

remain_dims[i] = false

Из 2x3x4 `remain_dims={true, false, true}`

получаем **3** коммуникатора с топологией **2x4**

Создание коммуникатора с топологией графа

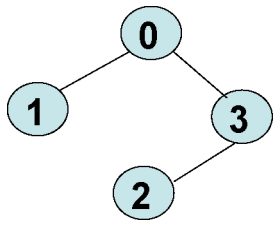
```
int MPI_Graph_create(
      MPI_Comm comm, int nnodes,
      int *index, int *edges,
      int reorder,
      MPI_Comm *comm_graph)
```

`index[nnodes]`- суммарное количество соседей для первых `i` вершин

`edges[index[nnodes-1]]`- упорядоченный список номеров процессов-соседей всех вершин

Создание коммуникатора с топологией графа (пример)

Процесс	Соседи
0	1, 3
1	0
2	3
3	0, 2




```
int index[4] = { 2, 3, 4, 6 };
int edges[6] = { 1, 3, 0, 3, 0, 2 };
MPI_Graph_create(MPI_COMM_WORLD, 4,
        index, edges, 1, &comm_graph);
```

Непосредственные соседи процесса

```
MPI_Graph_neighbors_count(
    MPI_Comm comm, int rank,
    int *neighbors_count)

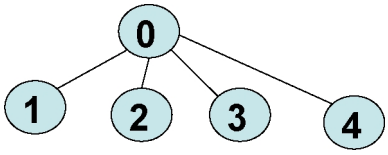
MPI_Graph_neighbors( MPI_Comm comm,
    int rank, int max,
    int *neighbors)
```

Параметры графа

```
MPI_Graphdims_get(MPI_Comm comm,
     int *nnodes, int *nedges)

MPI_Graph_get(MPI_Comm comm,
     int nnodes, int nedges,
     int *index, int * edges)
```

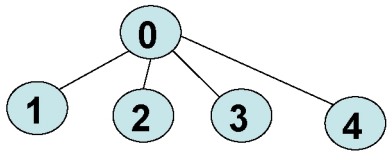
Пример создания и использования топологии типа «граф»



Общение процессов по коммуникационной схеме «master-slave»

```
program example_graph
include 'mpif.h'
integer ierr, rank, rank1, i, size, MAXPROC, MAXEDGES
parameter (MAXPROC = 128, MAXEDGES = 512)
integer a, b
integer status(MPI_STATUS_SIZE)
integer comm_graph, index(MAXPROC), edges(MAXEDGES)
integer num, neighbors(MAXPROC)
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
do i = 1, size
index(i) = size+i-2
end do
do i = 1, size-1
edges(i) = i
edges(size+i-1) = 0
end do
```

```
size = 5
do i = 1, size
index(i) = size+i-2
end do
```



index	4	5	6	7	8			
edges								

```
do i = 1, size-1
edges(i) = i
edges(size+i-1) = 0
end do
```

index	4	5	6	7	8			
edges	1	2	3	4	0	0	0	0

```
call MPI_GRAPH_CREATE(MPI_COMM_WORLD, size, index,
& edges, .TRUE., comm_graph, ierr)

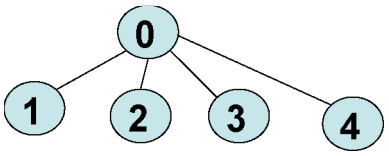
call MPI_GRAPH_NEIGHBORS_COUNT(comm_graph, rank,
& num, ierr)
call MPI_GRAPH_NEIGHBORS(comm_graph, rank, num,
& neighbors, ierr)

do i = 1, num
call MPI_SENDRECV(rank, 1, MPI_INTEGER, neighbors(i),
& 1, rank1, 1, MPI_INTEGER, neighbors(i), 1, comm_graph,
& status, ierr)
print *, 'procecc ', rank, ' communicate with process', rank1
end do

call MPI_FINALIZE(ierr)
end
```

```
call MPI_GRAPH_NEIGHBORS_COUNT(comm_graph, rank,
& num, ierr)
call MPI_GRAPH_NEIGHBORS(comm_graph, rank, num,
& neighbors, ierr)
```

rank	0	1	2	3	4
num	4	1	1	1	1
neighbors	1	0	0	0	0
	2				
	3				
	4				




```
do i = 1, num
call MPI_SENDRECV(rank, 1, MPI_INTEGER, neighbors(i),
& 1, rank1, 1, MPI_INTEGER, neighbors(i), 1, comm_graph,
& status, ierr)
end do
```

```
rank = 0
```

```
call MPI_SENDRECV(rank, 1, MPI_INTEGER, 1, 1,
& rank1, 1, MPI_INTEGER, 1, 1, comm_graph, status, ierr)

call MPI_SENDRECV(rank, 1, MPI_INTEGER, 2, 1,
& rank1, 1, MPI_INTEGER, 2, 1, comm_graph, status, ierr)

call MPI_SENDRECV(rank, 1, MPI_INTEGER, 3, 1,
& rank1, 1, MPI_INTEGER, 3, 1, comm_graph, status, ierr)

call MPI_SENDRECV(rank, 1, MPI_INTEGER, 4, 1,
& rank1, 1, MPI_INTEGER, 4, 1, comm_graph, status, ierr)
```

```
 rank = 1
```

call MPI_SENDRECV(rank, 1, MPI_INTEGER, 0, 1,
& rank1, 1, MPI_INTEGER, 0, 1, comm_graph, status, ierr)

```
 rank = 2
```

call MPI_SENDRECV(rank, 1, MPI_INTEGER, 0, 1,
& rank1, 1, MPI_INTEGER, 0, 1, comm_graph, status, ierr)

```
 rank = 3
```

call MPI_SENDRECV(rank, 1, MPI_INTEGER, 0, 1,
& rank1, 1, MPI_INTEGER, 0, 1, comm_graph, status, ierr)

```
 rank = 4
```

call MPI_SENDRECV(rank, 1, MPI_INTEGER, 0, 1,
& rank1, 1, MPI_INTEGER, 0, 1, comm_graph, status, ierr)

```
procecc 0 communicate with 1
procecc 0 communicate with 2
procecc 0 communicate with 3
procecc 0 communicate with 4
procecc 1 communicate with 0
procecc 2 communicate with 0
procecc 3 communicate with 0
procecc 4 communicate with 0
```

Библиотека MPI

Пример

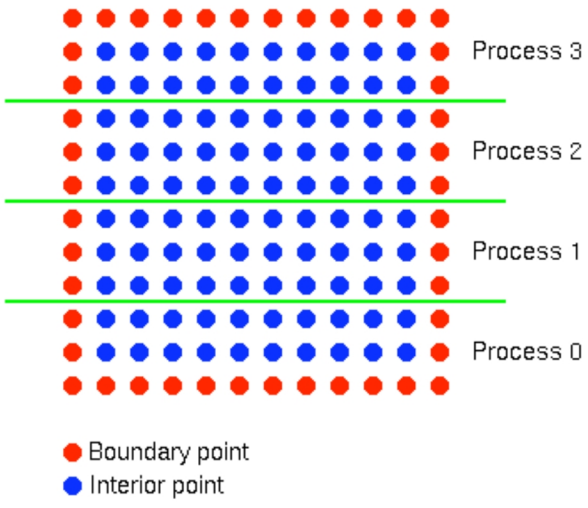
Решение уравнения Лапласа методом Якоби

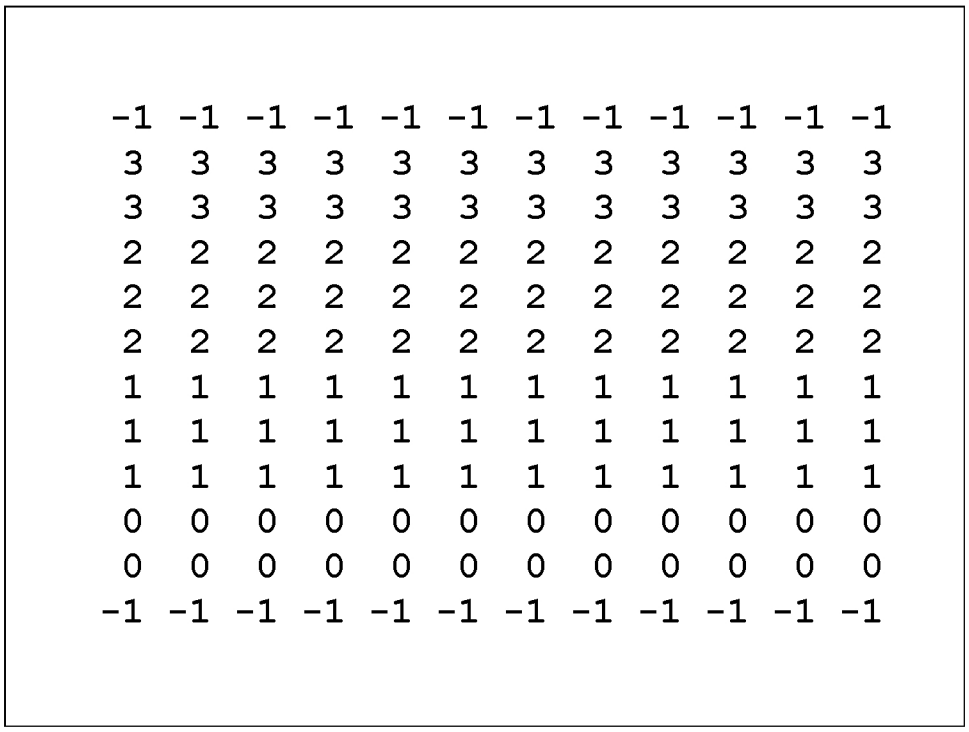
```
while (not converged) {
  for (i,j)
    xnew[i][j] = ( x[i+1][j]
                 + x[i-1][j]
                 + x[i][j+1]
                 + x[i][j-1])/4;
  for (i,j)
    x[i][j] = xnew[i][j];
  }
```

```
diffnorm = 0;

for (i,j)
    diffnorm += (xnew[i][j] - x[i][j])
              * (xnew[i][j] - x[i][j]);

diffnorm = sqrt(diffnorm);
```






```
#include <stdio.h>
#include <math.h>
#include "mpi.h"
#define maxn 12
int main( argc, argv )
int argc;
char **argv;
{
    int rank, value, size, errcnt, toterr, itcnt;
    int i, j, i_first, i_last;
    MPI_Status status;
    double     diffnorm, gdiffnorm;
    double     xlocal[(12/4)+2][12];
    double     xnew[(12/4)+2][12];
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    if (size != 4) MPI_Abort( MPI_COMM_WORLD, 1 )
```

```
/* xlocal[0][] is lower ghostpoints,
   xlocal[maxn+2][] is upper */
/* Note that top and bottom processes
   have one less row of interior points */
    i_first = 1;
    i_last  = maxn/size;
    if (rank == 0)        i_first++;
    if (rank == size - 1) i_last--;

    /* Fill the data as specified */
    for (i=1; i<=maxn/size; i++)
     for (j=0; j<maxn; j++)
         xlocal[i][j] = rank;
    for (j=0; j<maxn; j++) {
     xlocal[i_first-1][j] = -1;
     xlocal[i_last+1][j] = -1;
    }
```

```
for (i=1; i<=maxn/size; i++)
     for (j=0; j<maxn; j++)
         xlocal[i][j] = rank;
```

rank = 0	0	
	1	0 0 0 …
i_first = 2	2	0 0 0 …
i_last = 3	3	0 0 0 …
	4	

rank = 1	0	
i_first = 1	1	1 1 1 …
	2	1 1 1 …
i_last = 3	3	1 1 1 …
	4	

rank = 2	0	
i_first = 1	1	2 2 2 …
	2	2 2 2 …
i_last = 3	3	2 2 2 …
	4	

rank = 3	0	
i_first = 1	1	3 3 3 …
i_last = 2	2	3 3 3 …
	3	3 3 3 …
	4	

```
for (j=0; j<maxn; j++) {
     xlocal[i_first-1][j] = -1;
     xlocal[i_last+1][j] = -1;
```

rank = 0	0	
	1	-1 -1 …
i_first = 2	2	0 0 0 …
i_last = 3	3	0 0 0 …
	4	-1 -1 …

rank = 1	0	-1 -1 …
i_first = 1	1	1 1 1 …
	2	1 1 1 …
i_last = 3	3	1 1 1 …
	4	-1 -1 …

rank = 2	0	-1 -1 …
i_first = 1	1	2 2 2 …
	2	2 2 2 …
i_last = 3	3	2 2 2 …
	4	-1 -1 …

rank = 3	0	
i_first = 1	1	3 3 3 …
i_last = 2	2	3 3 3 …
	3	-1 -1 …
	4	

```
    itcnt = 0;
    do {









    } while (gdiffnorm > 1.0e-2 && itcnt < 100);
    MPI_Finalize( );
    return 0;
}
```

```
/* Send up unless I'm at the top,
   then receive from below
  Note the use of xlocal[i] for &xlocal[i][0] */

if (rank < size - 1)
    MPI_Send( xlocal[maxn/size], maxn, MPI_DOUBLE,
          rank + 1, 0, MPI_COMM_WORLD );
if (rank > 0)
    MPI_Recv( xlocal[0], maxn, MPI_DOUBLE,
          rank - 1, 0, MPI_COMM_WORLD, &status );
/* Send down unless I'm at the bottom */
if (rank > 0)
    MPI_Send( xlocal[1], maxn, MPI_DOUBLE,
          rank - 1, 1, MPI_COMM_WORLD );
if (rank < size - 1)
    MPI_Recv( xlocal[maxn/size+1], maxn,
          MPI_DOUBLE, rank + 1, 1,
          MPI_COMM_WORLD, &status );
```

```
if (rank < size - 1)
    MPI_Send( xlocal[maxn/size], maxn, MPI_DOUBLE,
          rank + 1, 0, MPI_COMM_WORLD );
if (rank > 0)
    MPI_Recv( xlocal[0], maxn, MPI_DOUBLE,
          rank - 1, 0, MPI_COMM_WORLD, &status );
```

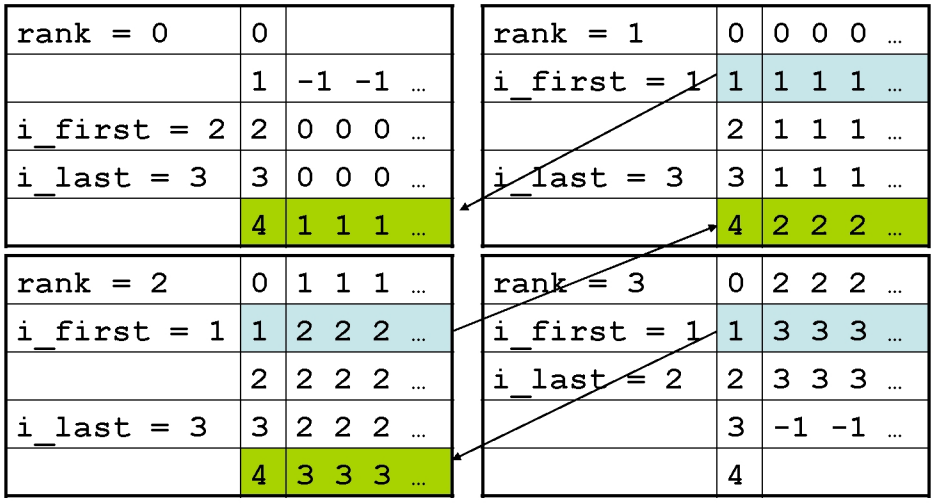
rank = 0	0	
	1	-1 -1 …
i_first = 2	2	0 0 0 …
i_last = 3	3	0 0 0 …
	4	-1 -1 …

rank = 1	0	0 0 0 …
i_first = 1	1	1 1 1 …
	2	1 1 1 …
i_last = 3	3	1 1 1 …
	4	-1 -1 …

rank = 2	0	1 1 1 …
i_first = 1	1	2 2 2 …
	2	2 2 2 …
i_last = 3	3	2 2 2 …
	4	-1 -1 …

rank = 3	0	2 2 2 …
i_first = 1	1	3 3 3 …
i_last = 2	2	3 3 3 …
	3	-1 -1 …
	4	

```
if (rank > 0)
    MPI_Send( xlocal[1], maxn, MPI_DOUBLE,
              rank - 1, 1, MPI_COMM_WORLD );
if (rank < size - 1)
    MPI_Recv( xlocal[maxn/size+1], maxn,
      MPI_DOUBLE, rank + 1, 1,MPI_COMM_WORLD, &status );
```




```
/* Compute new values (but not on boundary) */
itcnt ++;
diffnorm = 0.0;
for (i=i_first; i<=i_last; i++)
    for (j=1; j<maxn-1; j++) {
     xnew[i][j] =  (  xlocal[i][j+1]
                    + xlocal[i][j-1]
                    + xlocal[i+1][j]
                    + xlocal[i-1][j] ) / 4.0;
     diffnorm += (xnew[i][j] - xlocal[i][j]) *
                 (xnew[i][j] - xlocal[i][j]);
    }
/* Only transfer the interior points */
for (i=i_first; i<=i_last; i++)
    for (j=1; j<maxn-1; j++)
     xlocal[i][j] = xnew[i][j];
```

```
MPI_Allreduce( &diffnorm, &gdiffnorm, 1,
     MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD );

gdiffnorm = sqrt( gdiffnorm );
if (rank == 0)
     printf( "At iteration %d, diff is %e\n",
               itcnt, gdiffnorm );
```

```
% mpicc -o jacobi jacobi.c -lm
% mpirun -np 4 jacobi
At iteration 1, diff is 3.791438e+00
At iteration 2, diff is 1.948557e+00
At iteration 3, diff is 1.324536e+00
At iteration 4, diff is 1.014272e+00
At iteration 5, diff is 8.266808e-01
At iteration 6, diff is 6.992270e-01
At iteration 7, diff is 6.063208e-01


At iteration 88, diff is 1.219929e-02
At iteration 89, diff is 1.170511e-02
At iteration 90, diff is 1.123095e-02
At iteration 91, diff is 1.077600e-02
At iteration 92, diff is 1.033949e-02
At iteration 93, diff is 9.920652e-03
```

Стандарт OpenMP

Разработка программ для
многопроцессорных систем
с общей памятью

Информационные ресурсы

- www.openmp.org
- Что такое OpenMP - http://parallel.ru/tech/tech_dev/openmp.html
- OpenMP C/C++ specification v1.0 http:// www.openmp.org
- Introduction to OpenMP - www.llnl.gov/computing/tutorials/workshops/workshop/openMP/MAIN.html
- Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J. (2000). *Parallel Programming in OpenMP.* Morgan Kaufmann Publishers.

Содержание

- Стратегия подхода
  - OpenMP - стандарт параллельного программирования для систем с общей памятью
  - Положительные стороны
  - Принципы организации параллелизма
  - Структура OpenMP
- Директивы OpenMP
  - Формат записи директив
  - Области видимости директив
  - Типы директив
  - Определение параллельной области
    - Директива **parallel**
  - Распределение вычислений между потоками
    - Директива **DO/for**
    - Директива **sections**
    - Директива **single**
- Информационные ресурсы

Содержание

- Директивы OpenMP
  - Синхронизация
    - Директивы **master**, **critical**, **barrier**, **atomic**, **flush**, **ordered**, **threadprivate**
  - Управление областью видимости данных
    - Параметры директив **shared, private, firstprivate, lastprivate, copyin, default, reduction**
  - Совместимость директив и их параметров
- Библиотека функций OpenMP
  - Функции для контроля/запроса параметров среды исполнения
  - Функции синхронизации
- Переменные среды
- Реализации OpenMP

Стратегия подхода…

Интерфейс OpenMP задуман как стандарт параллельного программирования для многопроцессорных систем с общей памятью (SMP, ccNUMA, …)



В общем вид системы с общей памятью описываются в виде модели параллельного компьютера с произвольным доступом к памяти (*parallel random-access machine – PRAM*)

Стратегия подхода…

Динамика развития стандарта

- OpenMP Fortran API v1.0 (1997)
- OpenMP C/C++ API v1.0 (1998)
- OpenMP Fortran API v2.0 (2000)
- (OpenMP C/C++ API v2.0 – ожидается в 2001)

Разработкой стандарта занимается организация OpenMP ARB (ARchitecture Board), в которую вошли представители крупнейших компаний - разработчиков SMP-архитектур и программного обеспечения.

Стратегия подхода…

При разработке стандарта в значительной степени учитывались результаты предшествующих исследований

- X3H5 (ANSI стандарт, 1993) - MPP
- POSIX Threads (*Ptheads*), IEEE 1003.1c, 1995 - Unix

Стратегия подхода…

Основания для достижения эффекта – разделяемые для параллельных процессов данные располагаются в общей памяти и для организации взаимодействия не требуется операций передачи сообщений.

Интерфейс OpenMP служит основой для формирования новой технологии параллельного программирования для систем общей памяти (*shared-memory programming*)

Стратегия подхода…

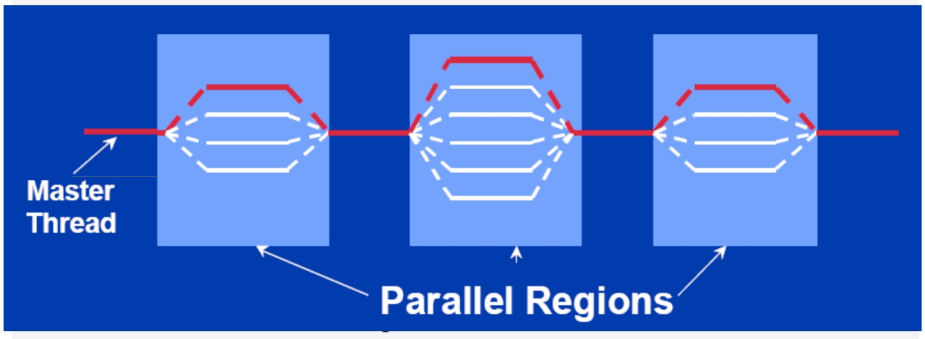
Положительные стороны

- Поэтапное (инкрементальное) распараллеливание
  - Можно распараллеливать последовательные программы поэтапно, не меняя их структуру
- Единственность разрабатываемого кода
  - Нет необходимости поддерживать последовательный и параллельный вариант программы, поскольку директивы игнорируются обычными компиляторами (в общем случае)
- Эффективность
  - Учет и использование возможностей систем с общей памятью
- Стандартизованность (переносимость), поддержка в наиболее распространенных языках (C, Fortran) и платформах (Windows, Unix)

Стратегия подхода…

Принцип организации параллелизма…

- Использование потоков (общее адресное пространство)
- Пульсирующий ("вилочный", *fork-join*) параллелизм



Стратегия подхода

Принцип организации параллелизма…

- При выполнении обычного кода (вне параллельных областей) программа выполняется одним потоком (master thread)
- При появлении директивы #parallel происходит создание “команды“ (team) потоков для параллельного выполнения вычислений
- После выхода из области действия директивы #parallel происходит синхронизация, все потоки, кроме master, уничтожаются
- Продолжается последовательное выполнение кода (до очередного появления директивы #parallel)

Стратегия подхода

Структура

- Набор директив компилятора
- Библиотека функций
- Набор переменных окружения

Изложение материала будет проводиться на примере языка C

Директивы OpenMP

Формат записи директив

Формат

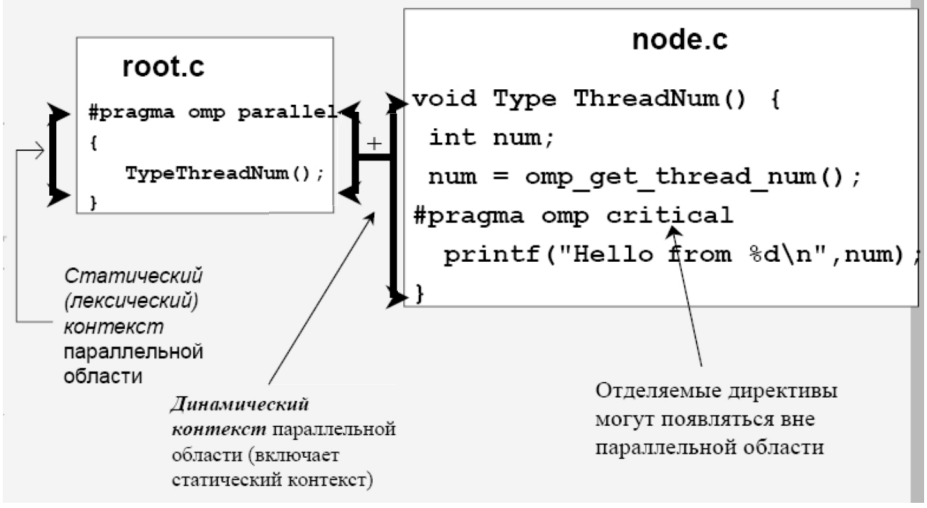
```
#pragma omp имя_директивы [clause,…]
```

Пример

```
#pragma omp parallel default(shared) \
                          private(beta,pi)
```

Директивы OpenMP

Области видимости директив



Директивы OpenMP

Типы директив

Директивы OpenMP можно разделить на 3 типа:

- определение параллельной области,
- разделение работы,
- синхронизация.

Директивы OpenMP

Определение параллельной области…

Директива **parallel** (основная директива OpenMP)

- Когда поток достигает директиву **parallel**, создается набор (*team*) из N потоков; входной поток является основным потоком этого набора (*master thread*) и имеет номер 0;
- Код области дублируется или разделяется между потоками для параллельного выполнения;
- В конце области обеспечивается синхронизация потоков – выполняется ожидание завершения вычислений всех потоков; далее все потоки завершаются – дальнейшие вычисления продолжает выполнять только основной поток.

Директивы OpenMP

Определение параллельной области…

Директива **parallel** (формат)

```
#pragma omp parallel [clause ...] newline
   structured_block
```

clause

```
   if (scalar_expression)
   private (list)
   shared (list)
   default (shared | none)
   firstprivate (list)
   reduction (operator: list)
   copyin (list)
```

Директивы OpenMP

Определение параллельной области…

Директива **parallel** (семантика)

- Количество потоков (по убыванию старшинства)
  - **omp_set_num_threads()**
  - **OMP_NUM_THREADS**
  - параметр реализации
- Динамические потоки
  - По умолчанию количество потоков одинаковое – для активизации динамического режима функция **omp_set_dynamic()** или переменная **OMP_DYNAMIC**
- Вложенность параллельных областей (во многих реализациях не обеспечивается)
  - по умолчанию во вложенной области создается один поток;
  - управление - функция **omp_set_nested()** или переменная **OMP_NESTED**
- Параметры (clause) – если условие в if не выполняется, то процессы не создаются

Директивы OpenMP

Определение параллельной области

Директива **parallel** (пример)

```
#include <omp.h>
main () {
 int nthreads, tid;
 /* Создание параллельной области */
#pragma omp parallel private(nthreads, tid)
 {
  /* печать номера потока */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);
  /* Печать количества потоков - только master */
  if (tid == 0) {
   nthreads = omp_get_num_threads();
   printf("Number of threads = %d\n", nthreads);
  }
 }/* Завершение параллельной области  */
}
```

Директивы OpenMP

Распределение вычислений между потоками…

Существует 3 директивы для распределения вычислений в параллельной области

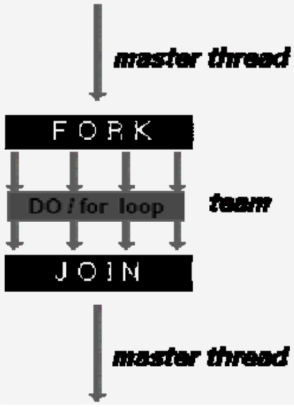
- **DO / for** – распараллеливание циклов
- **sections** – распараллеливание раздельных фрагментов кода (функциональное распараллеливание)
- **single** – директива для указания последовательного выполнения кода

! Начало выполнения директив по умолчанию не синхронизируется, завершение директив по умолчанию является синхронным

Директивы OpenMP

Распределение вычислений между потоками…

Директива **DO/for** – распределение итераций цикла между потоками; параллельная область уже должна быть активна, иначе игнорируется



Директивы OpenMP

Распределение вычислений между потоками…

Директива **DO/for** (формат)

```
#pragma omp for [clause ...] newline
  for_loop
```

clause

```
schedule (type [,chunk])
ordered
private (list)
firstprivate (list)
lastprivate (list)
shared (list)
reduction (operator: list)
nowait
```

Директивы OpenMP

Распределение вычислений между потоками…

Директива **DO/for** - распределение итераций регулируется параметром **schedule**

- **static** – итерации делятся на блоки по chunk итераций и статически разделяются между потоками; если параметр chunk не определен, итерации делятся между потоками равномерно и непрерывно
- **dynamic** – распределение итерационных блоков осуществляется динамически (по умолчанию chunk=1)
- **guided** – размер итерационного блока уменьшает экспоненциально при каждом распределении; chunk определяет минимальный размер блока (по умолчанию chunk=1)
- **runtime** – правило распределения определяется переменной **OMP_SCHEDULE** (при использовании runtime параметр chunk задаваться не должен)

Директивы OpenMP

Распределение вычислений между потоками…

Директива **DO/for** (пример)

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000
main () {
 int i, n, chunk;
 float a[NMAX], b[NMAX], c[NMAX];
 /* Some initializations */
 for (i=0; i < NMAX; i++)
  a[i] = b[i] = i * 1.0;
 n = NMAX;
 chunk = CHUNK;
 #pragma omp parallel shared(a,b,c,n,chunk) private(i)
 {
  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < n; i++)
    c[i] = a[i] + b[i];
 } /* end of parallel section */
}
```

Директивы OpenMP

Распределение вычислений между потоками…

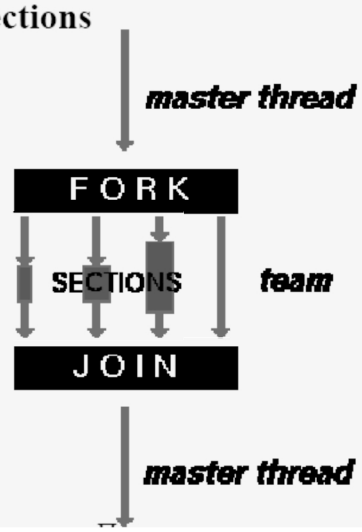
Директива **sections** – распределение вычислений для раздельных фрагментов кода

- фрагменты выделяются при помощи директивы **section**
- каждый фрагмент выполняется однократно
- разные фрагменты выполняются разными потоками
- завершение директивы по умолчанию синхронизируется
- директивы section должны использоваться только в статическом контексте

Директивы OpenMP

Распределение вычислений между потоками…

Директива **sections**



Директивы OpenMP

Распределение вычислений между потоками…

Директива **sections** (формат)

```
#pragma omp sections [clause ...] newline
{
 #pragma omp section newline
   structured_block
 #pragma omp section newline
   structured_block
}
clause
 private (list)
 firstprivate (list)
 lastprivate (list)
 reduction (operator: list)
 nowait
```

Директивы OpenMP

Распределение вычислений между потоками…

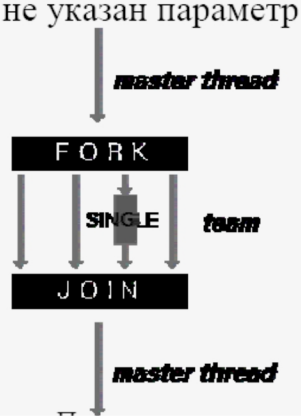
Директива **sections** (пример)

```
#include <omp.h>
#define NMAX 1000
main () {
 int i, n;
 float a[NMAX], b[NMAX], c[NMAX];
/* Some initializations */
 for (i=0; i < NMAX; i++)
   a[i] = b[i] = i * 1.0;
 n = NMAX;
 #pragma omp parallel shared(a,b,c,n) private(i)
 {
  #pragma omp sections nowait
  {
    #pragma omp section
      for (i=0; i < n/2; i++)
       c[i] = a[i] + b[i];
    #pragma omp section
      for (i=n/2; i < n; i++)
       c[i] = a[i] + b[i];
  } /* end of sections */
 } /* end of parallel section */
}
```

Директивы OpenMP

Распределение вычислений между потоками…

Директива **single** – определение фрагмента кода, который должен быть выполнен только одним потоком; все остальные потоки ожидают завершения выполнения фрагмента (если не указан параметр nowait)



Директивы OpenMP

Распределение вычислений между потоками…

Директива **single** (формат)

```
#pragma omp single [clause ...] newline
  structured_block

clause
 private (list)
 firstprivate (list)
 nowait
```

Директивы OpenMP

Распределение вычислений между потоками

Объединение директив **parallel** и **for/sections** в случае, если в параллельной области содержится только одну директиву **for** (**sections**)

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000
main () {
 int i, n, chunk;
 float a[NMAX], b[NMAX], c[NMAX];
 /* Some initializations */
 for (i=0; i < NMAX; i++)
  a[i] = b[i] = i * 1.0;
 n = NMAX;
 chunk = CHUNK;
 #pragma omp parallel for \
   shared(a,b,c,n) private(i) \
   schedule(static,chunk)
  for (i=0; i < n; i++)
    c[i] = a[i] + b[i];
}
```




Директивы OpenMP

Синхронизация…

Директива **master** определяет фрагмент кода, который должен быть выполнен только основным потоком; все остальные потоки пропускают данный фрагмент кода (завершение директивы по умолчанию не синхронизируется)

```
#pragma omp master newline
  structured_block
```

Директивы OpenMP

Синхронизация…

Директива **critical** определяет фрагмент кода, который должен выполняться только одним потоком в каждый текущий момент времени (*критическая секция*) основным потоком; все остальные потоки пропускают данный фрагмент кода (завершение директивы по умолчанию не синхронизируется)

```
#pragma omp critical [ name ] newline
  structured_block
```

Директивы OpenMP

Синхронизация…

Директива **critical** (пример)

```
#include <omp.h>
main() {
 int x;
 x = 0;
 #pragma omp parallel shared(x)
 {
  #pragma omp critical
    x = x + 1;
 } /* end of parallel section */
}
```

Директивы OpenMP

Синхронизация…

Директива **barrier** – определяет точку синхронизации, которую должны достигнуть все процессы для продолжения вычислений (директива должны быть вложена в блок)

```
#pragma omp barrier newline
```

Директивы OpenMP

Синхронизация…

Директива **atomic** – определяет переменную, доступ к которой (чтение/запись) должна быть выполнена как неделимая операция

```
#pragma omp atomic newline
   statement_expression
```

- Возможный формат записи выражения
  x binop = expr* , *x*++, ++*x*, *x*--, --*x
- ***x*** должна быть скалярной переменной
- ***expr*** не должно ссылаться на ***x***
- ***binop*** должна быть неперегруженной операцией вида
  +, -, *, /, &, ^, |, >>, <<

Директивы OpenMP

Синхронизация…

Директива **flush** – определяет точку синхронизации, в которой системой должно быть обеспечено единое для всех процессов состояние памяти (т.е. если потоком какое-либо значение извлекалось из памяти для модификации, измененное значение обязательно должно быть записано в общую память)

```
#pragma omp flush (list) newline
```

- Если указан список list, то восстанавливаются только указанные переменные
- Директива **flush** неявным образом присутствует в директивах barrier, critical, ordered, parallel, for, sections, single

Директивы OpenMP

Синхронизация…

Директива **ordered** – указывает фрагмент кода параллельного цикла, который должен выполняться точно в таком же порядке, как и при последовательном выполнении

```
#pragma omp ordered newline
  structured_block
```

- В каждый момент времени в блоке **ordered** может находиться только один поток
- На одной итерации цикла может быть только одна директива **ordered** и эта директива может выполниться только однократно
- Цикл, в котором имеется директива **ordered**, должен иметь параметр **ordered**

Директивы OpenMP

Синхронизация

Директива **threadprivate** – используется для создания поточных копий для глобальных переменных программы; созданные копии не видимы между потоками, но существуют во все время выполнения программы

```
#pragma omp threadprivate (list)
```

Директивы OpenMP

Управление областью видимости данных…

Общие (разделяемые между потоками, *shared*) переменные

– static, переменные с областью видимости в пределах файла

Локальные (*private*) данные потоков

- переменные циклы

Управление областью видимости обеспечивается при помощи параметров (*clause*) директив

private, firstprivate, lastprivate, shared, default, reduction, copyin

которые определяют, какие соотношения существуют между переменными последовательных и параллельных фрагментов выполняемой программы

Директивы OpenMP

Управление областью видимости данных…

Параметр **shared** определяет список переменных, которые будут общими для всех потоков параллельной области; правильность использования таких переменных должна обеспечиваться программистом

shared (list)

Параметр **private** определяет список переменных, которые будут локальными для каждого потока; переменные создаются в момент формирования потоков параллельной области; начальное значение переменных является неопределенным

private (list)

Директивы OpenMP

Управление областью видимости данных…

Параметр **firstprivate** позволяет создать локальные переменные потоков, которые перед использованием инициализируются значениями исходных переменных

firstprivate (list)

Параметр **lastprivate** позволяет создать локальные переменные потоков, значения которых запоминаются в исходных переменных после завершения параллельной области (используются значения потока, выполнившего последнюю итерацию цикла или последнюю секцию)

lastprivate (list)

Директивы OpenMP

Управление областью видимости данных…

Параметр **copyin** позволяет выполнить инициализацию переменных директивы threadprivate

copyin (list)

Параметр **default** устанавливает область видимости переменных по умолчанию

default (shared | none)

Директивы OpenMP

Управление областью видимости данных…

Параметр **reduction** определяет список переменных, для которых выполняется операция редукции; перед выполнением параллельной области для каждого потока создаются копии этих переменных, потоки формируют значения в своих локальных переменных и при завершении параллельной области на всеми локальными значениями выполняются необходимые операции редукции, результаты которых запоминаются в исходных (глобальных) переменных

reduction (operator: list)

Директивы OpenMP

Управление областью видимости данных…

Параметр **reduction** (правила записи)

- Возможный формат записи выражения
  - ***x = x op expr***
  - ***x = expr op x***
  - ***x binop = expr***
  - ***x++, ++x, x--, --x***
- ***x*** должна быть скалярной переменной
- ***expr*** не должно ссылаться на ***x***
- ***op*** (**operator**) должна быть неперегруженной операцией вида
  - **+, -, *, /, &, ^, |, &&, ||**
- ***binop*** должна быть неперегруженной операцией вида
  - **+, -, *, /, &, ^, |**

Директивы OpenMP

Управление областью видимости данных

Параметр **reduction** (пример)

```
#include <omp.h>
main () { /* vector dot product */
 int i, n, chunk;
 float a[100], b[100], result;
 /* Some initializations */
 n = 100; chunk = 10;
 result = 0.0;
 for (i=0; i < n; i++) {
  a[i] = i * 1.0; b[i] = i * 2.0;
 }
 #pragma omp parallel for \
   default(shared) private(i) \
   schedule(static,chunk) \
   reduction(+:result)
 for (i=0; i < n; i++)
  result = result + (a[i] * b[i]);
 printf("Final result= %f\n",result);
}
```

Директивы OpenMP

Совместимость директив и их параметров

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

Библиотека функций OpenMP

Функции для контроля/запроса параметров среды исполнения…

- `void omp_set_num_threads(int num_threads)`

  Позволяет назначить максимальное число потоков для использования в следующей параллельной области (если это число разрешено менять динамически). Вызывается из последовательной области программы.

- `int omp_get_max_threads(void)`

  Возвращает максимальное число потоков.

- `int omp_get_num_threads(void)`

  Возвращает фактическое число потоков в параллельной области программы.

Библиотека функций OpenMP

Функции для контроля/запроса параметров среды исполнения

Функции для контроля/запроса параметров среды исполнения

- `int omp_get_thread_num(void)`

  Возвращает номер потока.
- `int omp_get_num_procs(void)`

  Возвращает число процессоров, доступных приложению.
- `int omp_in_parallel(void)`

  Возвращает .TRUE., если вызвана из параллельной области программы.
- `void omp_set_dynamic(int dynamic)`
- `int omp_get_dynamic(void)`

  Устанавливает/запрашивает состояние флага, разрешающего динамически изменять число потоков.
- `void omp_get_nested(int nested)`
- `int omp_set_nested(void)`

  Устанавливает/запрашивает состояние флага, разрешающего вложенный параллелизм.

Библиотека функций OpenMP

Функции синхронизации…

В качестве замков используются общие переменные типа `omp_lock_t` или `omp_nestlock_t`.

Данные переменные должны использоваться только как параметры примитивов синхронизации.

```
void omp_init_lock(omp_lock_t *lock)
void omp_nest_init_lock(omp_nest_lock_t *lock)
```

Инициализирует замок, связанный с переменной `lock`

```
void omp_destroy_lock(omp_lock_t *lock)
void omp_destroy_nest_lock(omp_nest_lock_t
  *lock)
```

Удаляет замок, связанный с переменной lock.

Библиотека функций OpenMP

Функции синхронизации

- `void omp_set_lock(omp_lock_t *lock)`
- `void omp_set_nest__lock(omp_nest_lock_t *lock)`

  Заставляет вызвавший поток дождаться освобождения замка, а затем захватывает его.
- `void omp_unset_lock(omp_lock_t *lock)`
- `void omp_unset_nest__lock(omp_nest_lock_t *lock)`

  Освобождает замок, если он был захвачен потоком ранее.
- `void omp_test_lock(omp_lock_t *lock)`
- `void omp_test_nest__lock(omp_nest_lock_t *lock)`

  Пробует захватить указанный замок. Если это невозможно, возвращает .FALSE.

Переменные среды

`OMP_SCHEDULE`

Определяет способ распределения итераций в цикле, если в директиве DO использована клауза SCHEDULE(RUNTIME).

`OMP_NUM_THREADS`

Определяет число нитей для исполнения параллельных областей приложения.

`OMP_DYNAMIC`

Разрешает или запрещает динамическое изменение числа нитей.

`OMP_NESTED`

Разрешает или запрещает вложенный параллелизм.

Компилятор с поддержкой OpenMP определяет макрос "**_OPENMP**", который может использоваться для условной компиляции отдельных блоков, характерных для параллельной версии программы

Пример

```
#include <omp.h>
#define THREADNUMS 2
main () { /* вычисление числа π */
 long StepNums = 10000;
 double step, x, pi, sum=0.0;
 int i;
 step = 1.0/StepNums;
 omp_set_num_threads(THREADNUMS);
#pragma omp parallel for reduction(+:sun) private(i,x)
 for (i=0; i<StepNums; i++) {
   x=(i-0.5)*step;
   sum = sum + 4.0/(1.0-x*x);
 }
 pi = step * sum;
}
```

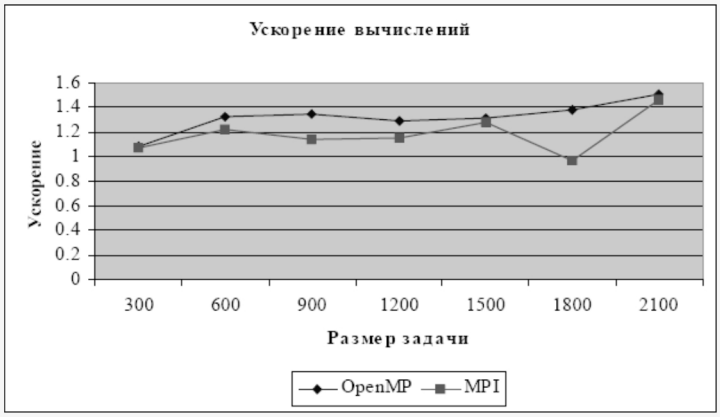
Сравнение технологий MPI и OpenMP для систем с общей памятью…

Сравнение времени выполнения последовательного варианта программы для задачи матричного умножения с вариантами OpenMP и MPI для 2-процессорного сервера

Порядок матрицы: (N)	Время $T_{посл}$ (последовательный алгоритм)	OpenMP		MPI	
		Время $T_{пар}$	Ускорение S	Время T	Ускорение S
300	0.42	0.39	1.08	0.39	1.07
600	4.69	3.55	1.32	3.85	1.22
900	16.20	12.05	1.34	14.17	1.14
1200	38.67	30.00	1.29	33.72	1.15
1500	76.56	58.20	1.32	60.19	1.27
1800	150.08	108.42	1.38	154.73	0.97
2100	258.09	171.75	1.50	177.03	1.46

Сравнение технологий MPI и OpenMP для систем с общей памятью…

Ускорение матричного умножения при использовании параллельных вычислений



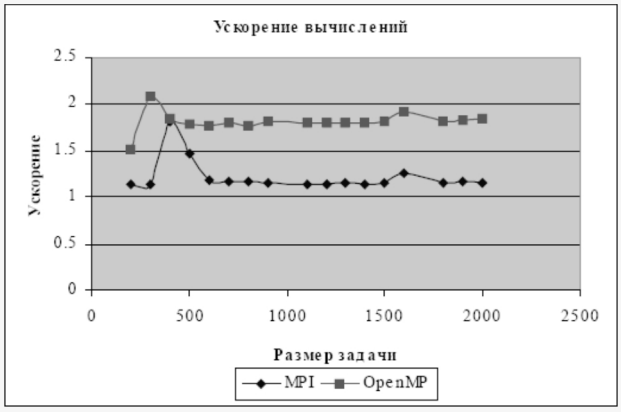
Сравнение технологий MPI и OpenMP для систем с общей памятью…

Сравнение времени выполнения последовательного варианта программы для задачи матричного умножения с вариантами OpenMP и MPI для 4-процессорного сервера

Порядок матрицы: (N)	Время $T_{посл}$ (последовательный алгоритм)	OpenMP		MPI	
		Время $T_{пар}$	Ускорение S	Время T	Ускорение S
300	0.36	0.17	2.09	0.32	1.14
600	6.66	3.78	1.76	5.61	1.19
900	22.92	12.70	1.80	20.01	1.15
1200	54.53	30.30	1.80	48.19	1.13
1500	107.91	59.67	1.81	93.27	1.16
1800	188.61	103.81	1.82	164.45	1.15
2000	262.09	142.73	1.84	226.45	1.16

Сравнение технологий MPI и OpenMP для систем с общей памятью

Ускорение матричного умножения при использовании параллельных вычислений



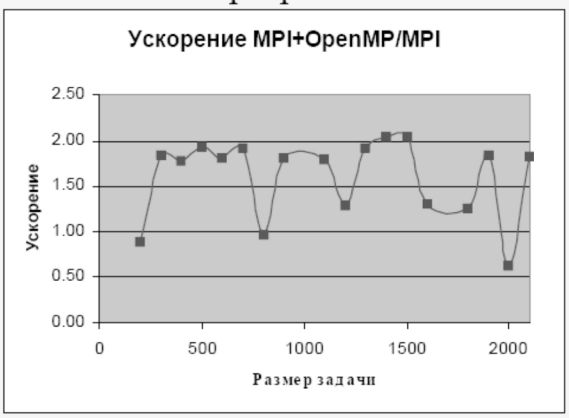
Комбинированная (MPI+OpenMP) технология программирования для систем с общей памятью…

Сравнение времени выполнения MPI и MPI+OpenMP вариантов программы для двух 2-процессорных серверов

Порядок матрицы: (N)	Время $T_{пар}$ (сек)		Ускорение S
	MPI	MPI+OpenMP	
300	0.52	0.28	1.84
600	3.95	2.18	1.82
900	14.03	7.75	1.81
1200	37.36	29.08	1.28
1500	65.39	32.03	2.04
1800	112.75	89.29	1.26
2100	185.85	101.97	1.82

Комбинированная (MPI+OpenMP) технология программирования для систем с общей памятью

Ускорение матричного умножения при использовании смешанного MPI+OpenMP варианта параллельной программы



Реализации OpenMP

1) **Silicon Graphics**. Fortran 77/90 (IRIX), планируется поддержка OpenMP для C/C++.
2) **Compaq/DEC**. **DIGITAL Fortran**
3) **Kuck & Associates** (KAI). Fortran, C/C++ (Unix, Windows)
4) **Portland Group** (PGI). Fortran и C/C++ для Windows NT, Linux, Solaris (x86).
5) **OdinMP**. OpenMP-препроцессор для языка C, генерация программы в стандарте POSIX threads.

Реализации OpenMP

6) **Sun.** Планируется поддержка OpenMP
7) **Pacific-Sierra Research** предлагает распараллеливающие препроцессоры **VAST/Parallel** для Fortran и C, которые обеспечивают автоматическое распознавание параллелизма в программах и выполнение необходимой трансформации программ путем добавления соответствующих директив OpenMP.
8) **Intel**. Fortran, C/C++ (Unix, Windows)

Программная система PVM (Parallel Virtual Machine)

Литература

- ***PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*** *(авторы: Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam). The MIT Press. Cambridge, Massachusetts. London, England.*

- http://www.netlib.org/pvm3/book/pvm-book.html

Литература

- Сервер Oak Ridge National Laboratory
- http://www.epm.ornl.gov/pvm/

Особенности организации параллельных вычислений

- Все задачи запускаются одной командой, в которой указывается имя запускаемого исполняемого файла, число запускаемых задач, а также число и тип используемых процессоров.
  По этой команде запускаются на указанных процессорах требуемое число копий указанного исполняемого файла.

Особенности организации параллельных вычислений

- Сначала запускается одна задача (master), которая в коллективе задач будет играть функции координатора работ. Эта задача производит некоторые подготовительные действия, после чего запускает остальные задачи (slaves), которым может соответствовать либо тот же исполняемый файл, либо разные исполняемые файлы.

Взаимодействие задач в PVM

- Каждая задача, запущенная на некотором процессоре, идентифицируется целым числом, которое называется идентификатором задачи - TID (по смыслу похоже на идентификатор процесса в операционной системе Unix)
- В пределах одной PVM любая задача может передавать сообщения любой другой задаче, причем, размеры и число таких сообщений в принципе не ограничены

Взаимодействие задач в PVM

- При передаче последовательности сообщение от одной задачи к другой порядок приема сообщение всегда совпадает с порядком их передачи
- Если до обращения к функции "Принять сообщение" в приемный буфер принимающей задачи записано несколько сообщений, то функция "Принять сообщение" возвратит ссылку на первое принятое сообщение

Взаимодействие задач в PVM

- Память для буферных массивов на передающей и приемной стороне выделяется динамически, следовательно, максимальный объем сообщений ограничивается только объемом доступной памяти
- Если одна из задач, запущенных в PVM, не может получить требуемую память для обмена, то выдается пользователю сообщение об ошибке ("cannot get memory"), но другие задачи об этом событии не извещаются и могут, например, продолжать посылать ей сообщения.

Взаимодействие задач в PVM

- Широковещательная передача сообщений от одной задачи к нескольким другим задачам
- Синхронизация действий в группе задач
- Совместное выполнение задачами группы некоторых операций над распределенными данными (фрагменты обрабатываемых данных распределены между задачами - членами группы)

Управление задачами

int tid = pvm_mytid (void);

- возвращает идентификатор задачи *tid >= 0*

Управление задачами

```
int numt = pvm_spawn (
   char *task,  /* имя исполняемого файла */
   char **argv, /* аргументы командной строки */
   int  flag,   /* опции запуска              */
   char *where, /* указывает место запуска    */
   int  ntask,  /* число запускаемых копий программы */
   int  *tids   /* массив значений tid для запущенных
                   задач */
);
```

запускает в PVM *ntask* копий исполняемого файла с именем *"task"* с одинаковыми аргументами командной строки в массиве *argv* и возвращает число запущенных задач *numt* а также последовательность идентификаторов для запущенных задач

Опции запуска

```
PvmTaskDefault ( 0 ) - Pvm выбирает по умолчанию, где
                       запустить задачи;
PvmTaskHost    ( 1 ) - задачи запускаются на машине,
                       тип которой указан в параметре
                       where;
PvmTaskArch    ( 2 ) - задачи запускаются на
                       комплексе, архитектура которого
                       указана в параметре where;
PvmTaskDebug   ( 4 ) - задачи стартуют под отладчиком;
PvmTaskTrace   ( 8 ) - при выполнении генерируются
                       результаты трассировки;
PvmMppFront    ( 16 ) - задачи стартуют на MPP системе;
PvmHostCompl   ( 32 ) - дополнение к информации в
                       параметре where.
```

Примеры запуска

```
numt = pvm_spawn( "host", 0,
                  PvmTaskHost, sparky",
                  2, &tid[0] );

numt = pvm_spawn ( "node", argv,
                  PvmTaskArch + PvmTaskDebug,"RIOS",
                  2, &tid[5] );

numt = pvm_spawn ( "node", 0, 0, 0, 4, tids );
```

Завершение задач

```
int  info = pvm_kill ( int  tid );
```

- завершает выполнение задачи с идентификатором *tid*;
  при ошибке возвращает код ошибки *info < 0*

```
int  info = pvm_exit ( void );
```

- завершает работу PVM, запущенной пользователем, но при этом сама задача продолжает выполняться уже как объект локальной операционной системы и завершает работу, как обычно

Обмен сообщениями между задачами

- При посылке сообщения:
  - проинициализировать буфер для передаваемого сообщения
  - упаковать передаваемые данные в соответствии с форматами их представления
  - переслать сообщение в буфер получателя

Обмен сообщениями между задачами

- При приеме сообщения:
  - определить алгоритм приема (блокированный или неблокированный
  - принять сообщение
  - распаковать, т.е. восстановить переданные фрагменты данных в соответствии с их типами

Функции для работы с буферами

int bufid = pvm_initsend (int encoding);

- возвращает значение, равное либо идентификатору буфера (*bufid > 0*), либо коду ошибки (*bufid < 0*)

Функцию **`pvm_initsend()`** следует вызывать ***каждый раз*** перед упаковкой и передачей нового сообщения.

Функции для работы с буферами (активные/неактивные)

```
int  bufid = pvm_mkbuf ( int encoding );
int  info = pvm_freebuf ( int bufid );

int  bufid = getsbuf ( void );
int  bufid = getrbuf ( void );

int  oldbuf = pvm_setsbuf ( int bufid );
int  oldbuf = pvm_setrbuf ( int bufid );
```

Значения параметра `encoding`:

- **`PvmDataDefault`** - при кодировании данных в буфере автоматически учитываются различия в представлении данных на передающей и на приемной стороне;
- **`PvmDataRaw`** - данные записываются в буфер без кодирования, что удобно, если их представления на передающей и на приемной стороне одинаковы;
- **`PvmDataInPlace`** - данные будут упаковываться и передаваться в приемный буфер, минуя буфер посылающей задачи, в котором только хранится информация о формате передаваемых сообщений. Это бывает целесообразно, когда требуется передавать множество сообщений с одним и тем же сложным форматом.

Функции упаковки сообщений

```
int info = pvm_pkbyte   ( char   *xp, int nitem, int stride );

int info = pvm_pkcplx   ( float  *cp, int nitem, int stride );
int info = pvm_pkdcplx  ( double *zp, int nitem, int stride );
int info = pvm_pkdouble ( double *dp, int nitem, int stride );
int info = pvm_pkfloat  ( float  *fp, int nitem, int stride );

int info = pvm_pkint    ( int    *ip, int nitem, int stride );
int info = pvm_pkuint   ( unsigned int *ip, int nitem, int str);
int info = pvm_pkshort  ( short  *ip, int nitem, int stride );
int info = pvm_pkushort ( unsigned short *ip, int nitem, int s);
int info = pvm_pklong   ( long   *ip, int nitem, int stride );
int info = pvm_pkulong  ( unsigned long *ip, int nitem, int s );

int info = pvm_pkstr    ( char   *cp, int nitem, int stride );
```

Функции для передачи сообщений

```
int  info = pvm_send ( int tid, int msgtag );
```
присваивает сообщению идентификатор *msgtag* и посылает сообщение задаче с идентификатором *tid*

```
int info = pvm_mcast ( int *tids, int ntask,
  int msgtag );
```
присваивает сообщению идентификатор *msgtag* и посылает его задачам, идентификаторы которых перечислены в *ntask* элементах массива *tids*

Функции для передачи сообщений

```
int  info = pvm_psend ( int tid, int msgtag,
  void *vp, int cnt, int type );
```
упаковывает *cnt* элементов массива *vd* типа *type* в сообщение, присваивает ему идентификатор *msgtag* и посылает его задаче с идентификатором *tid*

Эта функция позволяет обходиться без функций упаковки, если сообщение содержит лишь один массив простого типа.

Для значений параметра type используются следующие символические имена: *PVM_STR, PVM_BYTE, PVM_SHORT, PVM_INT, PVM_LONG, PVM_USHORT, PVM_UINT, PVM_ULONG, PVM_FLOAT, PVM_DOUBLE, PVM_CPLX, PVM_DCPLX*.

Функции для приема сообщений

`int bufid = pvm_recv ( int tid, int msgtag );`
- осуществляет блокированный прием, т.е. ожидает поступления сообщения с идентификатором *msgtag* от задачи с идентификатором *tid*, после чего создает ***новый*** приемный буфер с идентификатором *bufid*, помещает в него принятое сообщение и завершает работу

Если *tid* = *-1*, то принимается любое сообщение с идентификатором *msgtag*, направленное к данной задаче.

Если *msgtag* = *-1*, то принимается любое сообщение, направленное к данной задаче задачей с идентификатором *tid*.

Если *tid* = *msgtag* = *-1*, то принимается любое сообщение от любой задачи, направленное к данной задаче.

Функции для приема сообщений

```
int  info = pvm_bufinfo ( int bufid,
     int *nbytes, int *msgtag, int *tid );
```

- возвращает значения *tid, msgtag* и длину в байтах *nbytes* для сообщения в буфере с идентификатором *bufid*.

Эту функцию целесообразно использовать в сочетании с приемом сообщений при неопределенных значениях *tid = -1 и/или msgtag = -1*

Функции для приема сообщений

```
int bufid = pvm_nrecv ( int tid, int msgtag );
```

- осуществляет неблокированный прием, т.е. при отсутствии сообщения завершает работу, возвращая значение *bufid = 0*

```
int bufid = pvm_probe ( int tid, int msgtag );
```

- проверяет, поступило ли ожидаемое сообщение. Если оно поступило, то возвращается идентификатор приемного буфера *bufid > 0*, если сообщение еще не поступило, то возвращается значение *bufid = 0*

Функции для приема сообщений

```
int  bufid = pvm_trecv ( int tid,
     int msgtag, struct timeval *tmout );
```

- осуществляет блокированный прием сообщения, если время его ожидания не больше величины *tmout*, в противном случае через время *tmout* функция возвращает значение *bufid = 0*.
В структуре типа *timeval* имеется два целочисленных поля, в которые записываются число секунд и миллисекунд.

Распаковка данных из принятого сообщения

```
char cp[5]; int np[3]; double dp[7];
string  s[];
```

Упаковка:

```
info = pvm_pkbyte   ( cp, 5, 1 );
info = pvm_pkint    ( np, 3, 1 );
info = pvm_pkdouble ( dp, 7, 1 );
info = pvm_pkstr    ( s );
```

Распаковка:

```
char cp[]; int np[]; double dp[];
string  s[];
info = pvm_upkbyte   ( cp, 5, 1 );
info = pvm_upkint    ( np, 3, 1 );
info = pvm_upkdouble ( dp, 7, 1 );
info = pvm_upkstr    ( s );
```

Групповые функции

позволяют любому подмножеству выполняемых задач

- объединиться в группу, которая идентифицируется по имени, заранее известному ее членам
- организовать внутри группы взаимодействие задач:
  - синхронизация действий
  - обмен сообщениями
  - совместное выполнение глобальных операций

Групповые функции

```
int inum = pvm_joingroup(char *group);
```

- присоединяет задачу к группе *"group"* и присваивает ей порядковый номер в этой группе, равный *inum >= 0*. При ошибке *inum < 0*.

```
int  info = pvm_lvgroup ( char *group );
```

- выводит задачу из группы *"group"*.
При ошибке *info < 0*.

Групповые функции

```
int tid = pvm_gettid ( char *group, int inum );
```
- возвращает идентификатор задачи, которая в группе *"group"* имеет номер *inum*. При ошибке *tid < 0*.

```
int inum = pvm_getinst ( char *group, int tid );
```
- возвращает для задачи с идентификатором *tid* ее порядковый номер в группе *"group"*. При ошибке *inum < 0*.

```
int  size = pvm_gsize ( char *group );
```
- возвращает число задач в группе *"group"*. При ошибке *size < 0*.

Эти три функции могут вызываться ***любыми*** задачами в PVM, а не только членами группы *"group"*.

Групповые функции

- Функция для синхронизации задач
  int info = pvm_barrier (char *group, int count);
  - задача блокируется до того момента, когда функция *pvm_barrier()* будет вызвана ровно count участниками группы *"group"* (включая данную задачу).

- Функция передачи сообщений в группе
  int info = pvm_bcast (char *group, int msgtag);
  - посылает всем членам группы *"group"* (кроме самой себя) сообщение с идентификатором *msgtag*.

Групповые функции

- Функция для распределенных вычислений

  int info = pvm_reduce (void (*func)(), void *data, int item, int datatype, int msgtag, char *group, int root);

  - выполняет глобальную операцию, определяемую функцией ***func()***, с участием членов группы ***"group"*** и помещает результат операции в задачу, для которой номер в группе равен ***root***. Глобальная операция совершается над данными в массиве ***data***, тип которого указан в параметре ***datatype***. Длина массива ***data*** равна ***item***, и результат глобальной операции является массивом длины ***item***, в котором j-й элемент получен применением глобальной операции к j-м элементам массивов ***data***, размещенных в задачах, входящих в группу ***"group"***.

Символические имена значений параметра *datatype:*
PVM_BYTE,
PVM_SHORT, PVM_INT, PVM_LONG,
PVM_FLOAT, PVM_DOUBLE, PVM_CPLX,
PVM_DCPLX

В качестве первого параметра функции ***pvm_reduce()*** может быть либо указана пользовательская функция, либо подставлено имя одной из уже реализованных функций:

- **PvmMin** - вычисление минимального элемента
- **PvmMax** - вычисление максимального элемента
- **PvmSum** - вычисление суммы
- **PvmProduct** - вычисление произведения.

Т-система

Технология автоматического
динамического
распараллеливания программ

- использование парадигмы функционального программирования для обеспечения динамического распараллеливания программ

преимущества Т-системы

Для задач в которых:

- Априорно (до начала счета) неизвестно, как распараллелить работу
- Вычислительная схема близка к функциональной модели, то есть может быть эффективно представлена с помощью совокупности функций, рекурсивно вызывающих друг друга

- введение в императивный язык программирования (например, в язык С ++) понятие неготового значения
- вводится дополнительное ключевое слово (например, **tval**) в качестве необязательного атрибута переменных
- функциональная семантика легко и ненавязчиво для программиста проникает в программный код

- **tfun**— T-функции, то есть функции без побочных эффектов, вызовы которых можно вычислять параллельно
- **tout**— выход T-функции (аргумент, для возвращения посчитанного значения)

```
tfun void main(…) {
int w, t, a=5;
tval int x, y, z, b = 1;
G(a, b, &x, &y, &z);
b = z;
y = a;
w = a;
t = b + a *w;
…
}
```

```
001 #include <txx>
002
003 #include <stdio.h>
004 #include <stdlib.h>
005
007 int cfib (int n) {
009 return n < 2 ? n : cfib(n-1) + cfib(n-2);
010 }
011
012 tfun int fib (unsigned n)
013 {
014 if (n < 32) {
015 return cfib(n);
016 } else {
017 return fib(n-1) + fib(n-2);
021 }
022 }
023
```

```
024 tfun int main (int argc, char* argv[])
025 {
026 int n;
028 if (argc < 2) {
029 fprintf(stderr,"Usage: %s <number>\n", argv[0]);
032 return -1;
034 }
035 n = atoi(argv[1]);
038 printf("fib(%d) = %d\n", n, (int)fib(n));
039 return 0;
040 }
```

- В момент вызова **fib** соответствующая переменная становится «неготовой». Она содержит специальное *неготовое значение*. В дальнейшем это неготовое значение будет заменено обычным (готовым) значением: а именно в тот момент, когда вызванная функция **fib** посчитает и вернет в переменную свой результат

- Только что порожденный вызов Т-функций **fib** является новой гранулой параллелизма, готовой к вычислению. Она помещается в очередь, откуда локальные вычислительные процессы-исполнители (работающие на каждом вычислительном узле кластера, в количестве, равном числу процессоров в этом SMP узле) черпают работу сначала для себя, а затем (в случае полной локальной загрузки) и для других свободных вычислительных процессов в кластере

- Обращение к неготовым переменным на чтение (за значением) блокирует процесс вычисления функции
- Неготовые переменные, таким образом, являются средством синхронизации и посредниками при создании новых узлов редуцируемого вычислительного графа.
- Существенно различаются ситуации обращения к неготовым переменным на чтение (доступ за значением) и на запись (доступ для присваивания):

- *при чтении* происходит блокировка процесса вычисления, осуществившего такое обращение, и ожидание, когда переменная обретет готовое значение
- *при записи* обычного значения в неготовую переменную она становится готовой для всех потребителей ее результата, а ранее заблокированные на данной переменной процессы— разблокируются