ЯЗЫКИ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

Люди – это параллельные миры,
а реальная жизнь – лишь тонкая
поверхность их пересечения.
О. Муравьева

Языки программирования

Распараллеливающие компиляторы, которых нет

3

```
do 10 i=1, n
 do 10 j=1, n
  A(i+j)=A(2*n-i-j+1)*q+p
```

```
do 10 i=1, n
 A(i)=UserFunc(A, B(i))
```

```
do 10 i=1, n
 do 20 j=1, n-i
  A(i+j)=A(2*n-i-j+1)*q+p
 do 30 j=n-i+1, n
  A(i+j)=A(2*n-i-j+1)*q+p
```

?

Параллельные расширения существующих языков

4

- Дополнение имеющегося языка последовательного программирования параллельными конструкциями
  - Параллельные расширения и диалекты языка Fortran
    - Fortran-DVM, Cray MPP Fortran, F--, Fortran 90/95, Fortran D95, Fortran M, Fx, HPF, Opus, Vienna Fortran и др.
  - Параллельные расширения и диалекты языков C/C++
    - C-DVM, A++/P++, CC++, Charm/Charm++, Cilk, HPC, HPC++, Maisie, Mentat, mpC, MPC++, Parsec, pC++, sC++, uC++ и др.

Язык High Performance FORTRAN

5

- HPF – набор расширений языка FORTRAN, которые дают компилятору информацию для оптимизации выполнения программы на многопроцессорном (многоядерном) компьютере.
- Примеры
  - !HPF$ PROCESSORS (n)
    - Определение количества процессоров, которые могут использоваться программой.
  - !HPF$ DISTRIBUTE (BLOCK) ONTO procs :: список переменных
    - Блочное распределение данных массива список переменных по процессорам (распределение равными блоками).
  - ALIGN список переменных1(i) WITH список переменных2(i+1)
    - Установка связи между распределением двух массивов. Для всех значений переменной i элемент массива список переменных1(i) должен быть размещен в памяти того же процессора, что и элемент массива список переменных2(i).
  - FORALL (i=1:1000) список переменных1(i)=список переменных2(i)
    - Определение набора операторов, которые могут выполняться параллельно.

Параллельные языки и расширения

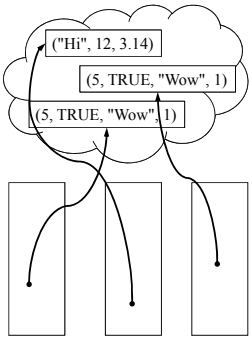
6

- Параллельные языки и расширения
  - HOPMA, ABCL, Adl, Ada, Concurrent Pascal, MC#, Erlang, Linda, Modula-3, NESL, occam, Orca, Parallaxis, Phantom, Sisal, SR, ZPL и др.

Язык и модель Linda

7

- Разработаны в 1980 гг. в Йельском университете (США).
- Параллельная программа – множество параллельных процессов, каждый из которых работает как последовательная программа.
- Процессы имеют доступ к общей памяти – *пространству кортежей*.
- *Кортеж* – упорядоченная последовательность значений.
- Процессы взаимодействуют друг с другом неявно, через пространство кортежей с помощью операций "поместить", "забрать", "скопировать" кортеж.
- Программа считается завершенной, если все процессы завершились или заблокированы.



Функции Linda

8

- OUT(<кортеж>)
  - Поместить кортеж в пространство кортежей.
  - Если такой кортеж уже имеется, то создается дубликат.
  - Вызывающий процесс не блокируется.
  - Примеры
    - out("myTuple", 1);
    - out(FALSE, 3.14,);

Функции Linda

9

- IN(<кортеж>)
  - Получить указанный кортеж и удалить его из пространства кортежей.
  - Если параметру соответствует несколько кортежей, то случайным образом выбирается один из них.
  - Вызывающий процесс блокируется, пока соответствующий кортеж не появится в пространстве кортежей.
  - Примеры
    - in("myTuple", 1);
    - int i, j;
      in("myTuple", 1, formal i, ?j);

Функции Linda

10

- READ(<кортеж>)
  - Получить указанный кортеж из пространства кортежей (не удаляя его).
  - Если параметру соответствует несколько кортежей, то случайным образом выбирается один из них.
  - Вызывающий процесс блокируется, пока соответствующий кортеж не появится в пространстве кортежей.
  - Примеры
    - int i;
      float j;
      char * s;
      read(?s, formal i, ?j);

Функции Linda

11

- EVAL(<кортеж>)
  - Поместить кортеж в пространство кортежей.
  - Если такой кортеж уже имеется, то создается дубликат.
  - Вызывающий процесс не блокируется.
  - Для вычисления поля кортежа, которое содержит обращение к какой-либо функции, порождается параллельный процесс.
  - Функция не ожидает завершения порожденного процесса.
  - Поля кортежа вычисляются в произвольном порядке.
  - Примеры
    - eval("myTuple", myFunc1(a,b,c), TRUE, myFunc1(i,j,k));

Примеры программ на языке Linda

12

- Получение номера собственного процесса и общего количества процессов
  - out("Next", 1);
  - in("Next", ?myNumber);
    out("Next", myNumber+1);
  - read("Next", ?numProcesses);

Примеры программ на языке Linda

13

- Барьерная синхронизация процессов
  - out("I want barrier", numProcesses);
  - in("I want barrier", ?Barrier);

```
Barrier--;
if (Barrier!=0) {
   out("I want barrier", Barrier);
   read("Barrier");
} else
   out("Barrier");
```

Примеры программ на языке Linda

14

- Параллельные процессы, работающие по схеме "мастер-рабочие"
  - int i, workers;
    void main(int argc, char * argv[])
    {
    workers=atoi(argv[1]);
    for (i=0; i<workers; i++)
    eval("Рабочий", Worker(i));
    for (i=0; i<workers; i++)
    in("Завершение работы");
    }
  - void Worker(int i)
    {
    /* Работа */
    out("Завершение работы");
    }

Параллельные API

15

- Программирование на стандартных языках программирования с использованием высокоуровневых коммуникационных библиотек и интерфейсов (API) для организации взаимодействия параллельных процессов (нитей).
- Коммуникационные библиотеки и интерфейсы
  - MPI, OpenMP, PVM, CVM, FM, Gala, GA, HPVM, ICC, Quarks, ROMIO, ShMem, SVMlib, TOOPS и др.

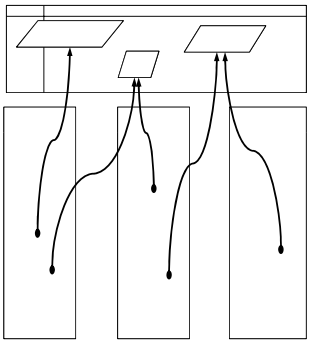
Технология OpenMP

16

- *OpenMP* – расширение языков C, C++ и FORTRAN, реализующее модель программирования в общей памяти и модель Fork-Join.
- Расширение представляет собой набор директив компилятора и спецификаций подпрограмм.
- Расширение реализуется разработчиками компиляторов для различных аппаратно-программных платформ.

Программирование в общей памяти

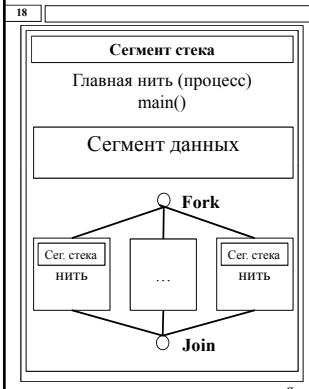
17



- Параллельное приложение состоит из нескольких процессов, выполняющихся одновременно.
- Процессы разделяют общую память.
- Обмены между процессами осуществляются посредством чтения/записи данных в общей памяти.

Модель FORK-JOIN

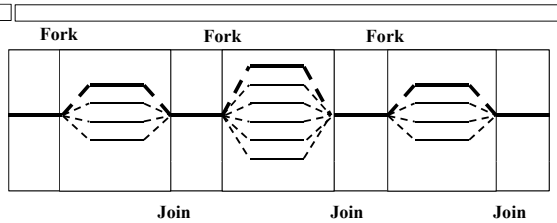
18



- Программа – полновесный *процесс (главная нить)*, который может запускать другие, *легковесные процессы (нити)*.
- Каждая нить имеет собственный сегмент стека.
- Все нити процесса разделяют сегмент данных процесса.

OpenMP-программа

19



- Последовательные регионы
- Параллельные регионы
- Нити
- Главная нить

Состав OpenMP

20

- *Переменные окружения* определяют поведение приложения, например:
  - OMP_NUM_THREADS – количество нитей в параллельном регионе
  - OMP_DYNAMIC – разрешение или запрет динамического изменения количества нитей.
  - OMP_NESTED – разрешение или запрет вложенных параллельных регионов.
- *Директивы компилятора #pragma* определяют поведение нитей, например:
  - #pragma omp parallel – создание параллельного региона
  - #pragma omp critical – определение критической секции.
- *Библиотечные функции* для просмотра и изменения параметров приложения, например:
  - int omp_get_thread_num(void) – номер текущей нити
  - int omp_get_num_procs(void) – общее количество нитей.

Простая программа на OpenMP

21

Последовательный код

```
void main()
{
  printf("Hello!\n");
}
```

Результат

```
Hello!
```

Параллельный код

```
#include "omp.h"
void main()
{
#pragma omp parallel
{
   printf("Hello!\n");
}
}
```

Результат (для 2-х нитей)

```
Hello!
Hello!
```


Область видимости переменных

22

- *Общая переменная (shared)* – доступна для модификации всем нитям.
- *Частная переменная (private)* – доступна для модификации только одной (создавшей ее) нити только на время выполнения этой нити.
- Правила видимости переменных:
  - все переменные, определенные **вне** параллельной области – **общие**;
  - все переменные, определенные **внутри** параллельной области – **частные**.

Общие и частные переменные

23

```
void main()
{
 int a, b, c;
 …
 #pragma omp parallel
 {
   int d, e;
   …
 }
}
```

- Общие переменные
  - a, b, c
- Частные переменные
  - d, e

Явное указание области видимости

24

- Для явного указания области видимости используются следующие параметры директив:
  - shared() – общие переменные
  - private() – частные переменные
- Примеры:

```
#pragma omp parallel shared(buf)
#pragma omp for private(i, j)
```


Общие и частные переменные

25

```
void main()
{
 int a, b, c;
 …
#pragma omp parallel shared(a) private(b)
{
   int d, e;
   …
}
}
```

- Общие переменные
  - a, c
- Частные переменные
  - b, d, e

Общие и частные переменные

26

```
void main()
{
  int rank;
 #pragma omp parallel
  {
    rank = omp_get_thread_num();
  }
  printf("%d\n", rank);
}
```

Одно (случайное) число из [0;OMP_NUM_THREADS-1]

```
void main()
{
  int rank;
 #pragma omp parallel
  {
    rank = omp_get_thread_num();
    printf("%d\n", rank);
  }
}
```

OMP_NUM_THREADS чисел из [0;OMP_NUM_THREADS-1] в случайном порядке, возможно, с повторениями

Общие и частные переменные

27

```
void main()
{
  int rank;
#pragma omp parallel
        shared (rank)
  {
    rank = omp_get_thread_num();
    printf("%d\n", rank);
  }
}
```

OMP_NUM_THREADS чисел из [0;OMP_NUM_THREADS-1] в случайном порядке, возможно, с повторениями

```
void main()
{
  int rank;
 #pragma omp parallel
        private (rank)
  {
    rank = omp_get_thread_num();
    printf("%d\n", rank);
  }
}
```

OMP_NUM_THREADS чисел из [0;OMP_NUM_THREADS-1] в случайном порядке без повторений

Распределение циклических вычислений между нитями

28

- #pragma omp for [список утверждений]
- Утверждения
  - private(список переменных)
  - firstprivate(список переменных)
  - lastprivate(список переменных)
  - reduction(оператор:список переменных)
  - ordered
  - schedule(тип распределения [,размер])
  - nowait

Распределение циклических вычислений между нитями

29

- #pragma omp for schedule(тип распределения [,размер])
- Типы распределения
  - static – итерации делятся на блоки заданного размера и статически разделяются между потоками; если размер не определен, итерации делятся между потоками равномерно и непрерывно
  - dynamic – распределение итерационных блоков осуществляется динамически (по умолчанию размер=1)
  - guided – размер итерационного блока уменьшается экспоненциально при каждом распределении; размер определяет минимальный размер блока (по умолчанию размер=1)
  - runtime – правило распределения определяется переменной OMP_SCHEDULE (при использовании runtime параметр размер задаваться не должен)

Распределение циклических вычислений между нитями

30

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000
main ()
{
  int i, n, chunk;
  float a[NMAX], b[NMAX], c[NMAX];
  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];
  }
}
```


Операция редукции

31

- Редукция подразумевает определение для каждой нити частной переменной для вычисления "частичного" результата и автоматическое выполнение операции "слияния" частичных результатов.

Операция	Нач. значение
+	0
*	1
-	0
^	0
&	~0
|	0
&&	1
\|\| | 0 |

Операция редукции

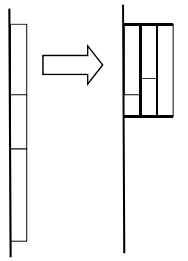
32

```
#include <omp.h>
void main ()
{
   int i, n, chunk;
   float a[100], b[100], result;
   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);
}
```


Явное распределение вычислений между нитями

33

```
#pragma omp parallel sections
{
   #pragma omp section
   phase1();
   #pragma omp section
   phase2();
   #pragma omp section
   phase3();
}
```



Послед. Паралл.

- Явное определение блоков кода, которые могут исполняться параллельно.

Явное распределение вычислений между нитями

34

- #pragma omp master
  - Определяет фрагмент кода, который должен быть выполнен только главной нитью (все остальные нити пропускают данный фрагмент).
- #pragma omp single
  - Определяет фрагмент кода, который должен быть выполнен только одной нитью – первой, достигнувшей данную точку (все остальные нити пропускают данный фрагмент).

Синхронизация нитей

35

- #pragma omp critical
  - Определяет критическую секцию – фрагмент кода, который должен выполняться только одной нитью в каждый текущий момент времени.
- #pragma omp barrier
  - Определяет точку барьерной синхронизации.

Технология OpenMP: резюме

36

- Поэтапное 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.
- Отсутствие передачи сообщений
  - Разделяемые нитями данные располагаются в общей памяти и для организации взаимодействия не требуется операций передачи сообщений.
- Мобильность
  - OpenMP фиксирован как стандарт, который реализован для языков C, C++, FORTRAN и ОС MS Windows и UNIX/Linux.

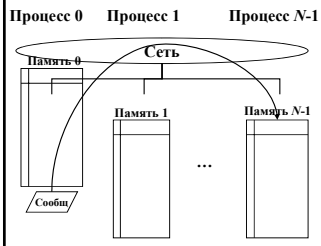
Технология MPI

37

- *MPI* – расширение языков C, C++ и FORTRAN, реализующее модель передачи сообщений и модели выполнения параллельных программ SPMD и MPMD.
- Расширение представляет собой набор спецификаций подпрограмм.
- Расширение реализуется разработчиками MPI-библиотек для различных аппаратно-программных платформ.

Модель передачи сообщений

38



- *Параллельное приложение* состоит из нескольких *процессов*, выполняющихся одновременно.
- Каждый процесс имеет приватную память.
- Обмены данными между процессами осуществляются посредством явной отправки/получения *сообщений*.
- Процессы могут выполняться как на одном и том же, так и на разных процессорах.

Модель выполнения SPMD

39

Процесс 0 Процесс 1 Процесс *N*-1

...

mytask.c mytask.c mytask.c

```
void main()
{
…
  switch (myrank) {
  case 0: do_mastertask(); /* Мастер */
  case 1: do_task1(); /* Рабочий 1 */
  case 2: do_task2();  /* Рабочий 2 */
   …
    }
…
}
```

- *Модель SPMD (Single Program Multiple Data)* предполагает, каждый процесс параллельного приложения имеет один и тот же исходный код.

Модель выполнения MPMD

40

Процесс 0 Процесс 1 Процесс *N*-1

...

master.c mytask1.c mytaskn.c

```
void
main()
{
 /*
  Мастер
*/
  …
}
```

```
void main()
{
 /*
  Рабочий 1
*/
  …
}
```

```
void main()
{
 /*
Рабочий N
*/
  …
}
```

- *Модель MPMD (Multiple Program Multiple Data)* предполагает, каждый процессы параллельного приложения имеют различные исходные коды.

MPI-программа

41

- *MPI-программа* – множество параллельных взаимодействующих процессов.
- Процессы порождаются один раз, во время запуска программы*.
- Каждый процесс работает в своем адресном пространстве, каких-либо общих данных нет. Единственный способ взаимодействия процессов – явный обмен сообщениями.

* Порождение дополнительных процессов и уничтожение существующих возможно только начиная с версии MPI-2.0.

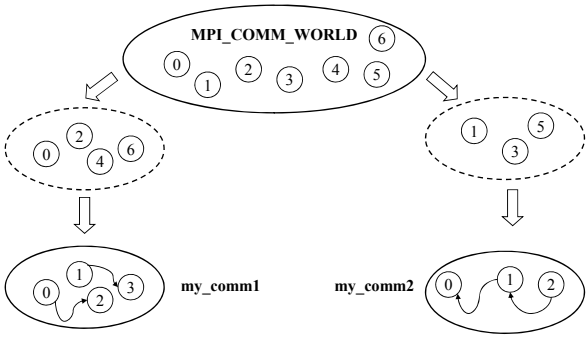
Коммуникаторы

42

- Для локализации области взаимодействия процессов можно создавать специальные программные объекты – *коммуникаторы*. Процесс может входить в разные коммуникаторы.
- Взаимодействия процессов проходят в рамках некоторого коммуникатора. Сообщения, переданные в разных коммуникаторах, не пересекаются и не мешают друг другу.
- Атрибуты процесса MPI-программы:
  - номер коммуникатора;
  - номер в коммуникаторе (от 0 до n-1, n – число процессов в коммуникаторе).
- Стандартные коммуникаторы:
  - MPI_COMM_WORLD – все процессы приложения
  - MPI_COMM_SELF – текущий процесс приложения
  - MPI_COMM_NULL – пустой коммуникатор

Коммуникаторы

43



Сообщение

44

- *Сообщение* процесса – набор данных стандартного (определенного в MPI) или пользовательского типа.
- Основные атрибуты сообщения:
  - номер процесса-отправителя (получателя)
  - номер коммуникатора
  - тег (уникальный идентификатор) сообщения (целое число)
  - тип элементов данных в сообщении
  - количество элементов данных
  - указатель на буфер с сообщением

Структура MPI-программы

45

```
#include "mpi.h"          /* Подключение библиотеки */

void main(int argc, char * argv[])
{

        MPI_Init(&argc, &argv); /* Инициализация */

        …                       /* Обмены  */

        MPI_Finalize();         /* Завершение */
}
```


MPI-функции

46

- Имеют имена вида MPI_...
- Возвращают целое число – MPI_SUCCESS или код ошибки.
- Простые функции общего назначения:
  - /* Количество процессов в коммуникаторе */
    int MPI_Comm_size(MPI_Comm comm, int * size);
  - /* Номер (ранг) текущего процесса в коммуникаторе */
    int MPI_Comm_rank(MPI_Comm comm, int * rank);

Пример MPI-программы

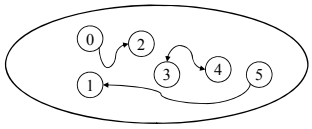
47

```
#include <stdio.h>
#include "mpi.h"
int total, iam;
int main(int argc, char * argv[])
{
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &total);
        MPI_Comm_rank(MPI_COMM_WORLD, &iam);
        printf("Привет! Я %d-й процесс из %d.\n", iam, total);
        MPI_Finalize();
        return 0;
}
```

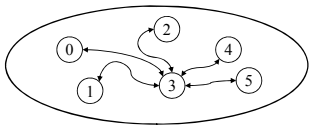

Виды взаимодействия процессов

48

- *Взаимодействие "точка-точка"* – обмен между двумя процессами одного коммуникатора.



- *Коллективное взаимодействие* – обмен между всеми процессами одного коммуникатора.



Взаимодействие "точка-точка"

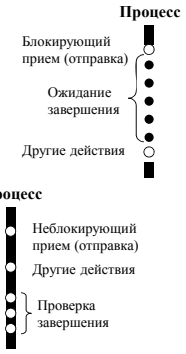
49

- Участвуют два процесса: отправитель сообщения и получатель сообщения.
  - Отправитель должен вызвать одну из функций отправки сообщения и явно указать атрибуты получателя (коммуникатор и номер в коммуникаторе) и тег сообщения.
  - Получатель должен вызвать одну из функций получения сообщения и указать (тот же) коммуникатор отправителя; получатель может не знать номер отправителя и тег сообщения.
- Свойства:
  - Сохранение порядка (если P0 передает P1 сообщения A и затем B, то P1 получит A, а затем B).
  - Гарантированное выполнение обмена (если P0 вызвал функцию отправки, а P1 вызвал функцию получения, то P1 получит сообщение от P0).

Виды коммуникационных функций "точка-точка"

50

- *Блокирующая* функция запускает операцию и возвращает управление процессу только после ее завершения.
  - После завершения допустима модификация отправленного (принятого) сообщения.
- *Неблокирующая* функция запускает операцию и возвращает управление процессу немедленно.
  - Факт завершения операции проверяется позднее с помощью другой функции.
  - До завершения операции недопустима модификация отправляемого (получаемого) сообщения.



Отправка сообщений при использовании функций "точка-точка"

51

- *Стандартная* – завершается сразу после отправки сообщения.
- *Синхронная* – завершается после приема подтверждения от адресата.
- *Буферизованная* – завершается, как только сообщение копируется в системный буфер для дальнейшей отправки.
- *"По готовности"* – начинается, если адресат инициализировал прием и завершается сразу после отправки.

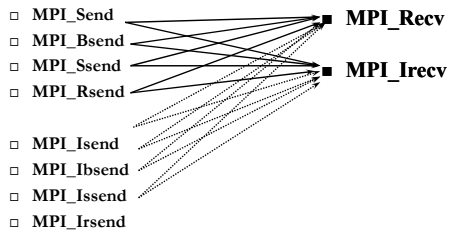
Коммуникационные функции "точка-точка"

- Отправка: **MPI_[I][R, S, B]Send**
- Прием: **MPI_[I]Recv**

Блокирующие			**Неблокирующие**		
Отправка		***Прием***	***Отправка***		***Прием***
Стандартная	**MPI_Send**	**MPI_Recv**	***Стандартная***	**MPI_Isend**	**MPI_Irecv**
Синхронная	**MPI_Ssend**		***Синхронная***	**MPI_Issend**	
Буферизованная	**MPI_Bsend**		***Буферизованная***	**MPI_Ibsend**	
По готовности	**MPI_Rsend**		***По готовности***	**MPI_Irsend**	

Коммуникационные функции "точка-точка"

53



Блокирующая стандартная отправка сообщения

54

- int MPI_Send
  - IN void * buf – указатель на буфер с сообщением
  - IN int count – количество элементов в буфере
  - IN MPI_Datatype datatype – MPI-тип данных элементов в буфере
  - IN int dest – номер процесса-получателя
  - IN int tag – тег сообщения
  - IN MPI_Comm comm – коммуникатор

Блокирующее стандартное получение сообщения

55

- int MPI_Recv
  - OUT void * buf – указатель на буфер с сообщением
  - IN int count – количество элементов в буфере
  - IN MPI_Datatype datatype – MPI-тип данных элементов в буфере
  - IN int src – номер процесса-отправителя
  - IN int tag – тег сообщения
  - IN MPI_Comm comm – коммуникатор
  - OUT MPI_Status* status – информация о фактически полученных данных (указатель на структуру с двумя полями: source –номер процесса-источника, tag – тег сообщения)

Неблокирующая стандартная отправка сообщения

56

- int MPI_Isend
  - IN void * buf – указатель на буфер с сообщением
  - IN int count – количество элементов в буфере
  - IN MPI_Datatype datatype – MPI-тип данных элементов в буфере
  - IN int dest – номер процесса-получателя
  - IN int tag – тег сообщения
  - IN MPI_Comm comm – коммуникатор
  - OUT MPI_Request *request – дескриптор операции (для последующей проверки завершения операции)

Неблокирующее стандартное получение сообщения

57

- int MPI_Irecv
  - OUT void * buf – указатель на буфер с сообщением
  - IN int count – количество элементов в буфере
  - IN MPI_Datatype datatype – MPI-тип данных элементов в буфере
  - IN int src – номер процесса-отправителя
  - IN int tag – тег сообщения
  - IN MPI_Comm comm – коммуникатор
  - OUT MPI_Request *request – дескриптор операции (для последующей проверки завершения операции)

Завершение неблокирующих обменов

58

- /* Проверка завершения */
  int MPI_Test
  (MPI_Request *request, int *flag, MPI_Status *status)
  - int MPI_Testany (…)
  - int MPI_Testall (…)
  - int MPI_Testsome (…)
- /* Ожидание завершения */
  int MPI_Wait
  (MPI_Request *request, MPI_Status *status)
  - int MPI_Waitany (…)
  - int MPI_Waitall (…)
  - int MPI_Waitsome (…)

Тупики (deadlocks)

59

- Гарантированный тупик

P0	P1
MPI_Recv (от процесса P1); MPI_Send (процессу P1);	MPI_Recv (от процесса P0); MPI_Send (процессу P0);

- Возможный тупик

P0	P1
MPI_Send (процессу P1); MPI_Recv (от процесса P1);	MPI_Send (процессу P0); MPI_Recv (от процесса P0);

Разрешение тупиков

60

P0	P1
MPI_Send (процессу P1); MPI_Recv (от процесса P1);	MPI_Recv (от процесса P0); MPI_Send (процессу P0);

P0	P1
MPI_Send (процессу P1); MPI_Recv (от процесса P1);	MPI_Irecv (от процесса P0); MPI_Send (процессу P0); MPI_Wait

P0	P1
/* Совмещенные прием и передача */ MPI_Sendrecv	/* Совмещенные прием и передача */ MPI_Sendrecv

Получение сообщений

61

- "Джокеры"
  - MPI_ANY_SOURCE – получить сообщение, отправленное любым процессом
  - MPI_ANY_TAG – получить сообщение, имеющее любой тег
- Информация об ожидаемом сообщении
  - /* С блокировкой */
    int MPI_Probe(int src, int tag, MPI_Comm comm, MPI_Status * status);
  - /* Без блокировки */
    int MPI_Iprobe
    (int src, int tag, MPI_Comm comm, int * flag, MPI_Status * status);
  - /* Количество элементов в принятом сообщении */
    int MPI_Get_count(MPI_Status * status, MPI_Datatype type, int * cnt);

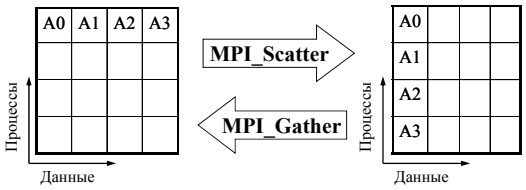
Коллективные операции

62

- Прием и/или передачу выполняют одновременно *все* процессы коммуникатора.
- Коллективная функция имеет большое количество параметров, часть которых нужна для приема, а часть для передачи. При вызове в разных процессах та или иная часть игнорируется.
- Значения *всех* параметров коллективных функций (за исключением адресов буферов) должны быть идентичными во всех процессах.
- MPI назначает теги для сообщений автоматически.

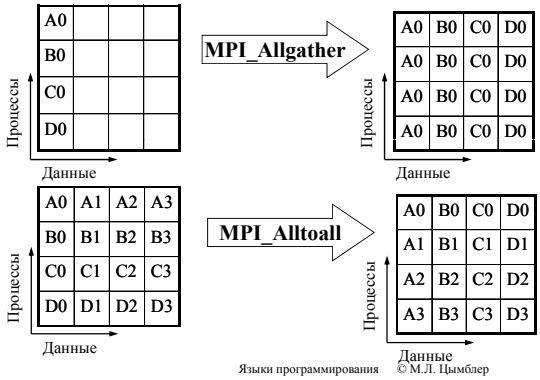
Коллективный прием сообщения

- /* Сборка элементов данных из буферов всех процессов в буфере процесса с номером rootRank */
  int MPI_Gather
  (void * sbuf, int scount, MPI_Datatype stype, void * rbuf,
  int rcount, MPI_Datatype rtype, int rootRank, MPI_Comm comm);
- /* Рассылка элементов данных из буфера процесса с номером rootRank в буфера всех процессов (обратная к MPI_Gather) */
  int MPI_Scatter



Широковещательные прием и передача

64



Глобальные операции над данными

65

- /* Выполнение count независимых глобальных операций op над соответствующими элементами массивов sbuf.
  Результат выполнения над *i*-ми элементами sbuf записывается в *i*-й элемент массива rbuf процесса rootRank. */
  int MPI_Reduce
  (void * sbuf, void * rbuf, int count, MPI_Datatype type, MPI_Op op, int rootRank, MPI_Comm comm);
- Глобальные операции:
  - MPI_MAX, MPI_MIN
  - MPI_SUM, MPI_PROD
  - …
  - MPI_Op_Create()

Стандартные типы данных в MPI

Тип MPI	Соответствующий тип C
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	-

Пользовательские типы данных

67

- /* Создание типа "массив" */
  int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype);

```
#define N 100
int A[N];
MPI_Datatype MPI_INTARRAY100;
…
MPI_Type_contiguous(N, MPI_INT, & MPI_INTARRAY100);
MPI_Type_commit(&MPI_INTARRAY100);
…
MPI_Send(A, 1, MPI_INTARRAY100, ... );
/* то же, что и MPI_Send(A, N, MPI_INT, ... ); */
…
MPI_Type_free( &intArray100 );
```