ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ В СТАНДАРТЕ MPI

Если у вас есть яблоко и у меня есть яблоко, и мы обмениваемся этими яблоками, то у вас и у меня остается по одному яблоку. А если у вас есть идея и у меня есть идея, и мы обмениваемся идеями, то у каждого из нас будет по две идеи.

Дж.Б. Шоу

Суперкомпьютеры и их применение

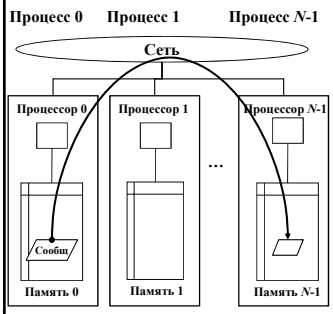
Содержание

2

- Модель передачи сообщений для параллельного программирования в системах с распределенной памятью
- Модели SPMD и MPMD запуска параллельных программ
- Стандарт Message Passing Interface (MPI)
- Основные понятия и функции MPI

Модель передачи сообщений

3



- *Параллельное приложение* состоит из нескольких *процессов*, выполняющихся одновременно.
- Каждый процесс имеет приватную память.
- Обмены данными между процессами осуществляются посредством явной отправки/получения *сообщений*.
- Процессы, как правило, выполняются на различных процессорах.

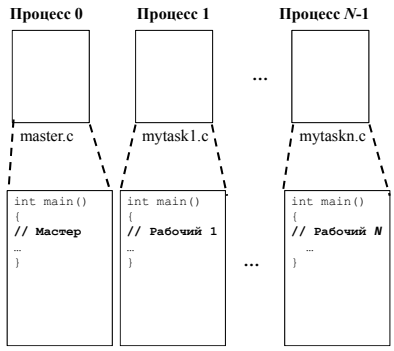
Модель SPMD

4

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

Модель MPMD

5



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

Стандарт MPI

6

- *MPI (Message Passing Interface)* – стандарт, реализующий модель обмена сообщениями между параллельными процессами. Поддерживает модели выполнения SPMD и (начиная с версии 2.0) MPMD.
- Стандарт представляет собой набор спецификаций подпрограмм (более 120) на языках C, C++ и FORTRAN.
- Стандарт реализуется разработчиками в виде библиотек подпрограмм для различных аппаратно-программных платформ (кластеры, персональные компьютеры, …, Windows, Unix/Linux, …).
- Коммерческие (IntelMPI, MSMPI и др.) и свободно распространяемые реализации стандарта (MPICH и др.).
- Текущая версия стандарта: http://www.mpi-forum.org.

Цикл разработки MPI-программ

7

- Персональный компьютер
  - Первоначальная разработка и отладка
  - Отладка по результатам тестирования на суперкомпьютере
- Суперкомпьютер
  - Тестирование
  - Эксперименты

MPI-программа

8

- *MPI-программа* – множество параллельных взаимодействующих процессов.
- Процессы порождаются один раз, в момент запуска программы средствами среды исполнения MPI программ*.
- Все процессы программы последовательно перенумерованы, начиная с 0. Номер процесса именуется *рангом* процесса.
- 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.

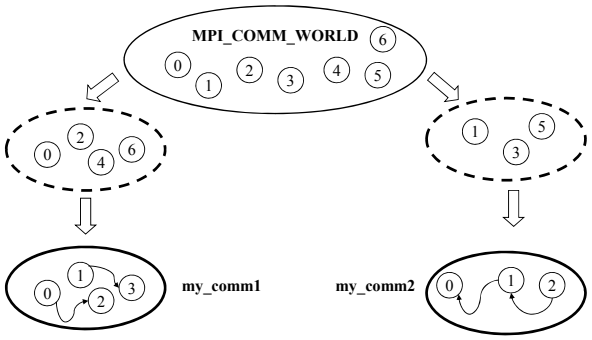
Коммуникаторы

9

- Для локализации области взаимодействия процессов можно динамически создавать специальные программные объекты – *коммуникаторы*. Один и тот же 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

10



Сообщение

11

- *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

12

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

int main (int argc, char *argv[])
{
// Здесь код без использования MPI функций

  MPI_Init(&agrc, &argv); // Инициализация выполнения

// Здесь код, реализующий обмены

  MPI_Finalize(); // Завершение

// Здесь код без использования MPI функций

 return 0;
}
```

MPI-функции

13

- 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) процесса в коммуникаторе`
    `int MPI_Comm_rank(MPI_Comm comm, int * rank);`
  - `// Замер времени (сек.)`
    `double MPI_Wtime(void);`
  - `// Имя текущего процессорного узла`
    `int MPI_Get_processor_name(char * name, int * len);`

Простая MPI-программа

14

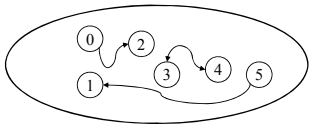
```
#include "mpi.h"
#include <stdio.h>

int main (int argc, char *argv[])
{
  int num, rank;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &num);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  printf("Привет! Я %d-й процесс из %d.\n", rank, num);
  MPI_Finalize();
  return 0;
}
```

```
Привет! Я 1-й процесс из 4.
Привет! Я 0-й процесс из 4.
Привет! Я 4-й процесс из 4.
Привет! Я 3-й процесс из 4.
Привет! Я 2-й процесс из 4.
```

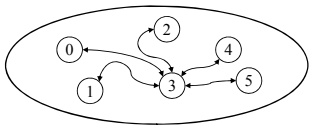

Виды взаимодействия процессов

15

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



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



Взаимодействие "точка-точка"

16

- 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/или 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"

17

- *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.

Блокирующие *vs* неблокирующие операции

18

- Блокирующие операции
  - Имеют более простую семантику.
  - Относительно легко используются в программе, не требуют дополнительных действий для завершения обмена.
  - Могут снизить быстродействие программы.
  - Могут привести к тупикам.
- Неблокирующие операции
  - Имеют более сложную семантику.
  - Относительно трудно используются в программе, требуют дополнительных действий для завершения обмена
  - Могут повысить быстродействие программы.
  - Не вызывают тупиков.

Режимы отправки сообщений "точка-точка"

19

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

Режимы отправки сообщений "точка-точка"

20

- Режим *по готовности* формально является наиболее быстрым, но используется достаточно редко, т.к. обычно сложно гарантировать готовность операции приема.
- *Стандартный* и *буферизованный* режимы также выполняются достаточно быстро, но могут приводить к большим расходам ресурсов (памяти), и могут быть рекомендованы для передачи коротких сообщений.
- *Синхронный* режим является наиболее медленным, т.к. требует подтверждения приема. В то же время этот режим наиболее надежен, и может быть рекомендован для передачи длинных сообщений.

Коммуникационные функции "точка-точка"

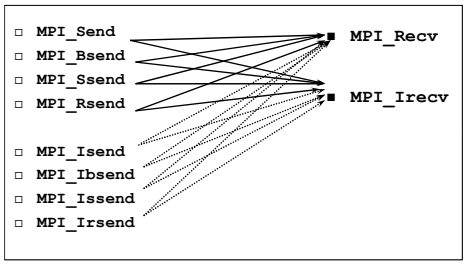
21

- 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"

22



Блокирующая стандартная отправка сообщения

23

- **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

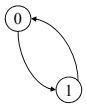
24

- **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)

Пример программы

25

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  int numtasks, rank, dest, src, rc, tag=777;
  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; src = 1;
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, src, tag, MPI_COMM_WORLD, &Stat);
} else
  if (rank == 1) {
    dest = 0; src = 0;
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, src, tag, MPI_COMM_WORLD, &Stat);
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }
  MPI_Finalize();
}
```



Неблокирующая стандартная отправка сообщения

26

- **`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

27

- **`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

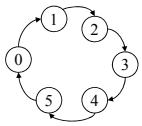
28

- 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 (…)

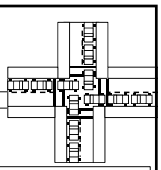
Пример

29

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  int numtasks, rank, next, prev, buf[2], tag1=111, tag2=222;
  MPI_Request reqs[4];
  MPI_Status stats[4];

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  prev = rank - 1;
  if (prev < 0) prev = numtasks - 1;
   next = rank + 1;
  if (next > numtasks - 1) next = 0;
  MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
  MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);
  MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
  MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);
  MPI_Waitall(4, reqs, stats);

  MPI_Finalize();
}
```



Тупики (deadlocks)



30

- 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

31

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

Совмещение приема и передачи сообщения

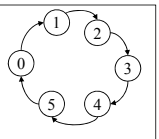
32

- **int MPI_Sendrecv**
  - OUT void * sbuf - адрес начала буфера с посылаемым сообщением;
  - IN int scount - число передаваемых элементов в сообщении;
  - IN MPI_Datatype stype - тип передаваемых элементов;
  - IN int dest - номер процесса-получателя;
  - IN int stag - идентификатор посылаемого сообщения;
  - OUT void *rbuf - адрес начала буфера приема сообщения;
  - IN int rcount - число принимаемых элементов сообщения;
  - IN MPI_Datatype rtype - тип принимаемых элементов;
  - IN int source - номер процесса-отправителя;
  - IN int rtag - идентификатор принимаемого сообщения;
  - IN MPI_Comm comm - идентификатор коммуникатора;
  - OUT MPI_Status status - параметры принятого сообщения.
- Объединяет в едином запросе посылку и прием сообщений и гарантирует отсутствие тупиков.
- Принимающий и отправляющий процессы могут являться одним и тем же процессом. Буфера приема и посылки обязательно должны быть различными. Сообщение, отправленное операцией MPI_Sendrecv, может быть принято обычным образом, и точно также операция MPI_Sendrecv может принять сообщение, отправленное обычной операцией MPI_Send.

Пример

33

```
#include "mpi.h"
#include <stdio.h>
int rank, numtasks, prev, next, buf[2];
MPI_Status status1, status2;

int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  prev=rank-1; if (prev < 0) prev=numtasks-1;
  next=rank+1; if (next > size-1) next=0;
  MPI_Sendrecv(rank, 1, MPI_INTEGER, prev, 777, buf[1], 1, MPI_INTEGER, next,
777, MPI_COMM_WORLD, status2);
  MPI_Sendrecv(rank, 1, MPI_INTEGER, next, 888, buf[0], 1, MPI_INTEGER, prev,
888, MPI_COMM_WORLD, status1);
  printf("Process %d, prev=%d, next=%d\n", rank, buf[0], buf[1]);

  MPI_Finalize();
  return 0;
}
```



Получение сообщений

34

- "Джокеры"
  - ранг процесса `MPI_ANY_SOURCE`
  - тег сообщения `MPI_ANY_TAG`
- 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

35

- 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.

Барьерная синхронизация

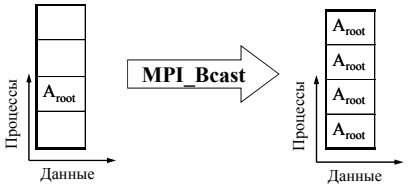
36

- `int MPI_Barrier(MPI_Comm comm)`
  - Обеспечивает *синхронизацию* процессов – одновременное достижение процессами указанной точки вычислений.
  - Должна вызываться всеми процессами коммуникатора.
  - Продолжение вычислений любого процесса произойдет по окончании выполнения функции MPI_Barrier всеми процессами коммуникатора.

Широковещательная рассылка

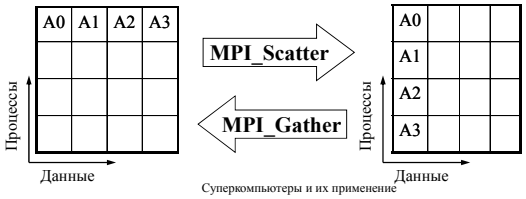
37

- int **MPI_Bcast**
  - IN/OUT void *buf – указатель на буфер с сообщением (отправляемым – для процесса с рангом root, принимаемым – для остальных процессов)
  - IN int count – количество элементов в буфере
  - IN MPI_Datatype datatype – MPI-тип данных элементов в буфере
  - IN int root – ранг процесса, выполняющего рассылку данных
  - IN MPI_Comm comm – коммуникатор



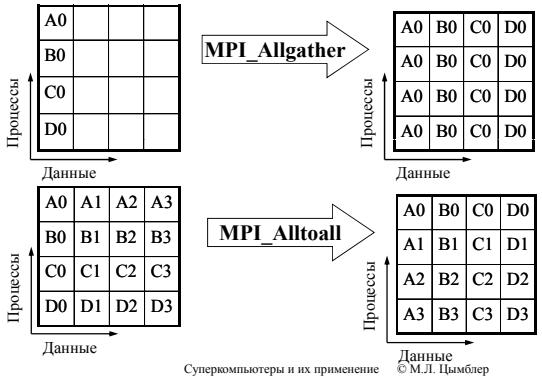
Коллективный прием сообщения

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



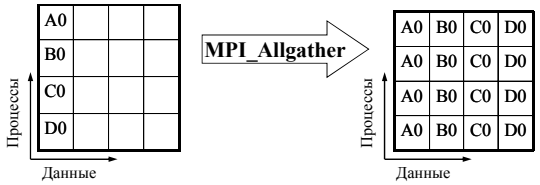
Широковещательные прием и передача

39



Широковещательный прием

40



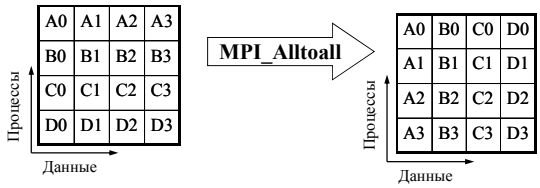

```
int MPI_Allgather(void *sbuf, int scount, MPI_Datatype stype,
    void *rbuf, int rcount, MPI_Datatype rtype, MPI_Comm comm)
```


Широковещательная передача

41

```
int MPI_Alltoall(void *sbuf, int scount, MPI_Datatype stype,
    void *rbuf, int rcount, MPI_Datatype rtype, MPI_Comm comm)
```

- **sbuf**, **scount**, **stype** - параметры передаваемых сообщений,
- **rbuf**, **rcount**, **rtype** - параметры принимаемых сообщений
- **comm** - коммуникатор, в рамках которого выполняется передача данных



Глобальные операции над данными

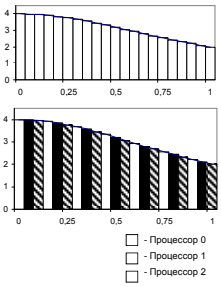
42

- 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 с рангом `root`.
  `int MPI_Reduce``(void * sbuf, void * rbuf, int count, MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm);`
- Глобальные операции:
  - `MPI_MAX, MPI_MIN`
  - `MPI_SUM, MPI_PROD`
  - ...
  - `MPI_Op_Create()`

Пример: вычисление π

43

- $\pi = \int_0^1 \frac{4}{1+x^2} dx$
- Метод прямоугольников для численного интегрирования.
- Циклическая схема распределения вычислений
  - Получаемые на отдельных процессорах частные суммы должны быть просуммированы.



Пример: вычисление π

44

```
#include "mpi.h"
#include <math.h>

int N; // Количество интервалов
double PI25DT=3.14159265358979323846462643;

double f(double a)
{
  return (4.0 / (1.0 + a*a));
}

int main(int argc, char *argv[])
{
  int rank, num, i;
  double mypi, pi, h, sum, x, t1, t2;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &num);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    printf("Number of intervals? "\n);
    scanf("%d\n", &N)
    t1 = MPI_Wtime();
  }
  MPI_Bcast(&N, 1, MPI_INT, 0,
MPI_COMM_WORLD);
```

```
  // Вычисление локальной суммы
  h = 1.0 / (double) N;
  sum = 0.0;
  for (i = rank + 1; i <= N; i += num) {
     x = h * ((double)i - 0.5);
     sum += f(x);
  }
  mypi = h * sum;

  // Сложение локальных сумм (редукция)
  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM,
0, MPI_COMM_WORLD);

// вывод результатов
  if (rank == 0) {
    t2 = MPI_Wtime();
    printf("Pi is %.16f,
            Error is %.16f\n",
            pi, fabs(pi - PI25DT));
    printf("Time is %f\n", t2-t1);
  }
  MPI_Finalize();
  return 0;
}
```


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

46

- 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 );
```


MPI-2: нововведения

47

- Динамическое порождение процессов
  - Разрешается создание и уничтожение процессов по ходу выполнения программы.
  - Предусмотрен специальный механизм, позволяющий устанавливать связь между только что порожденными процессами и уже работающей частью MPI-программы.
  - Имеется возможность установить связь между двумя приложениями даже в том случае, когда ни одно из них не было инициатором запуска другого.
- Одностороннее взаимодействие процессов
  - Обмен сообщениями по схеме Put/Get вместо традиционной схемы Send/Receive. Активной стороной может быть один процесс (при обмене не требуется активность отправителя либо получателя).
- Параллельный ввод-вывод
  - Специальный интерфейс для работы процессов с файловой системой.
- Расширенные коллективные операции
  - Во многие коллективные операции добавлена возможность взаимодействия между процессами, входящими в разные коммуникаторы.
  - Многие коллективные операции внутри коммуникатора могут выполняться в режиме, при котором входные и выходные буфера совпадают.
- Интерфейс для C++

Заключение

48

- Модель передачи сообщений для параллельного программирования в системах с распределенной памятью
- Модели SPMD и MPMD запуска параллельных программ
- Стандарт Message Passing Interface (MPI)
- Основные понятия и функции MPI