Архитектура компьютеров

ст. преп. кафедры МСС А.С. Гусейнова

Распределённые и параллельные вычисления.
Введение в MPI

Структура курса.

- 34 часа лекции
- 34 часа практики
- 34 часа практики:
  - Параллельные вычисления (Библиотека MPI)
  - Многопоточные программы(OpenMP)
  - Исследование объёма и производительности кэш-памяти разного уровня

Структура курса АК

- **Основы**, вводятся основные определения дисциплины и рассматриваются используемые инструменты.
- **Цифровая схемотехника (** рассматриваются способы построения различных функциональных блоков и узлов процессора)

Структура курса

- **архитектура и микроархитектура**, обсуждаются способы повышения производительности.
- **современные архитектуры процессорных систем** обсуждаются метрики эффективности и производительности.

Введение в MPI

- **Message Passing Interface** (MPI, интерфейс передачи сообщений) — программный интерфейс для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. Разработан Уильямом Гроуппом, Эвином Ласком и другими.

Высокопроизводительные вычисления (High Performance Computing)

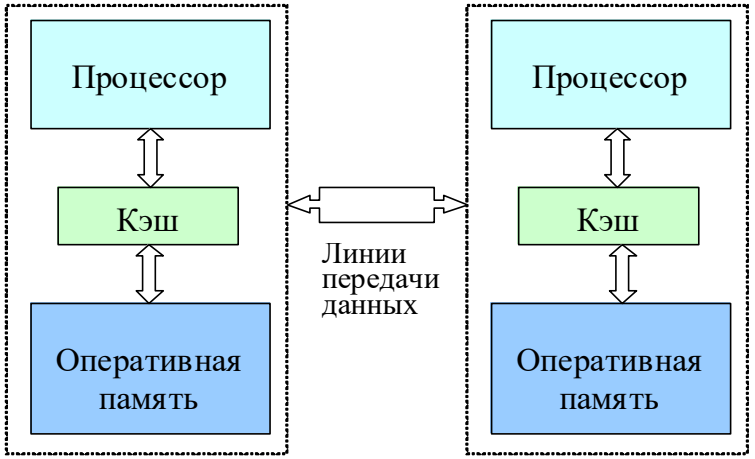
- Уменьшение времени сложных расчетов(научных)
- Расчеты связанные с большим объемом данных
- Задачи реального времени
- Надежные системы

План.

1. MPI: основные понятия и определения
2. Введение в MPI
   a) Инициализация и завершение MPI программ
   b) Определение количества и ранга процессов
   c) Прием и передача сообщений
   d) Определение времени выполнения MPI программы
3. Пример: первая программа с использованием MPI

Понятие MPI

MPI используется **в вычислительных системах с распределенной памятью**, в которых процессоры работают независимо друг от друга.



Для организации параллельных вычислений в таких системах необходимо уметь:

- *распределять* вычислительную нагрузку,
- *организовать* информационное взаимодействие (*передачу данных*) между процессорами.

Решение всех перечисленных вопросов обеспечивает ***MPI - интерфейс передачи данных*** *(message passing interface)*

Стандарт MPI

1993 г. – объединение нескольких групп в MPI Forum для создания единых требований к средствам программирования многопроцессорных систем с распределённой памятью .

Результат – стандарт MPI 1.0 в 1994 г.

Основные положения стандарта:

1. Реализации стандарта должны быть через подключаемые библиотеки или модули, без создания новых компиляторов или языков.
2. Библиотеки должны реализовывать все возможные типы обменов данными между процессорами (вычислительными узлами)

Стандарт MPI

MPI-4.0 был принят MPI Forum 9 июня, 2021

Скачать документацию стандарта можно по ссылке:

https://www.mpi-forum.org/docs/

Библиотека MPI

Существует для языков:

- Fortran
- ***C/C++***
- ***Java***
- ***.NET***

Представляет собой реализацию общих положений стандарта под тот или иной язык.

Мы рассматриваем реализацию **mpi.h** для C/C++

Библиотеки реализующие стандарт MPI

- **MPICH** (сокр. от англ. *«Message Passing Interface CHameleon»*) — одна из самых первых разработанных библиотек MPI. На её базе было создано большое количество других библиотек как открытых, так и коммерческих.
- Сторонние разработки, основанные на MPICH: Intel MPI, HP MPI, Microsoft MPI, IBM MPI, Cray MPI и др.

Библиотеки реализующие стандарт MPI

- Open MPI-свободная реализация MPI

https://www.open-mpi.org

Разработки основанные на Open MPI:

Oracle MPI

Высокоуровневые интерфейсы:

C++ (**Boost.MPI**)

Java (Open MPI Java Interface, MPI Java)

C# (MPI.NET, MS-MPI)

Python (MPI4py, PyMPI)

MPI-программа будет работать на любой многоядерной или многопроцессорной системе.

MPI поддерживается:

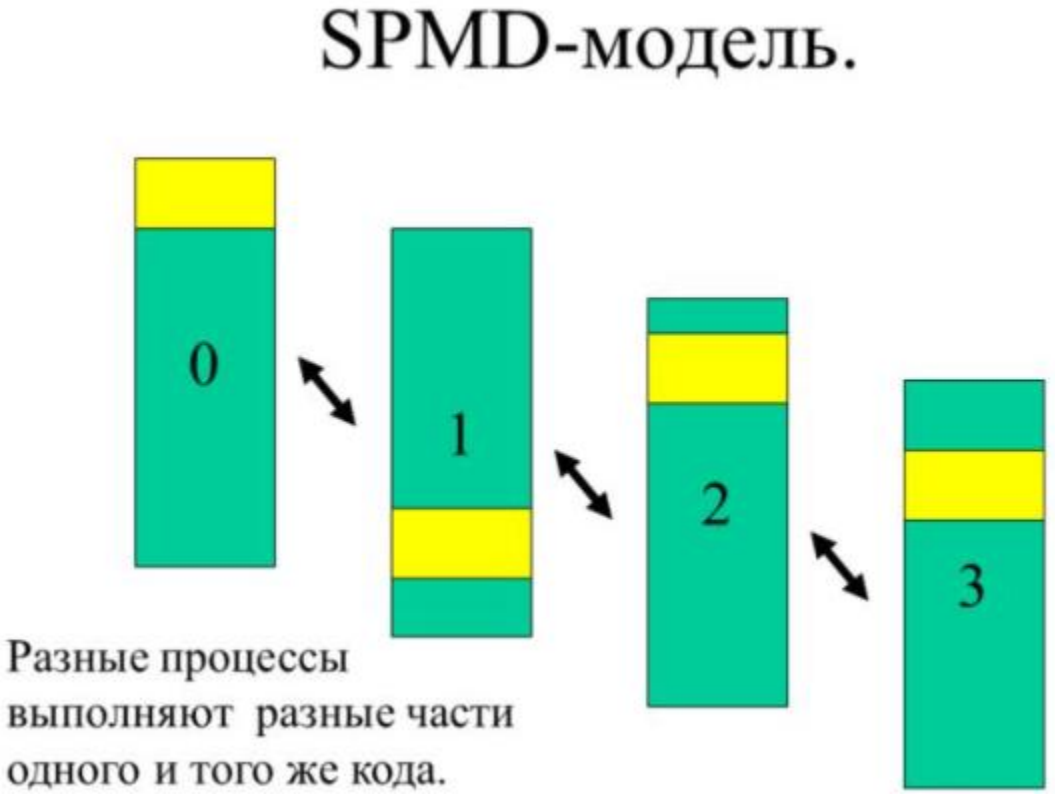
- на классических MPP-системах;
- на кластерах;
- в сетях рабочих станций;
- на SMP-системах.

Single program multiple processes or *SPMP*

- В рамках MPI для решения задачи разрабатывается **одна программа**, она запускается на выполнение **одновременно на всех** имеющихся процессорах
- Для организации различных вычислений на разных процессорах:
  - Есть возможность подставлять разные данные для программы на разных процессорах,
  - Имеются средства для идентификации процессора, на котором выполняется программа

Такой способ организации параллельных вычислений обычно именуется как *модель "одна программа множество процессов"* (*single program multiple processes* or *SPMP*)

Single program multiple processes or *SPMP*



MPI: основные понятия и определения…

Понятие параллельной программы

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

MPI: основные понятия и определения…

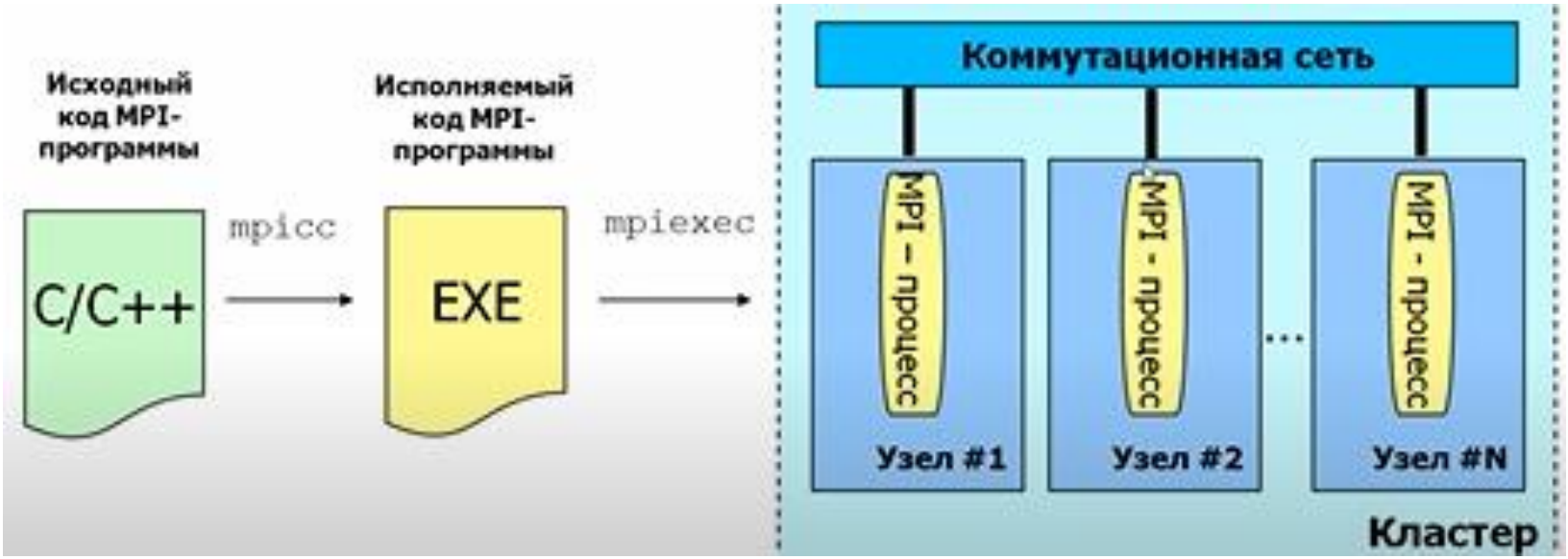
Количество процессов определяется *в момент запуска* параллельной *программы* средствами среды исполнения MPI программ.

Все процессы программы **последовательно перенумерованы.**

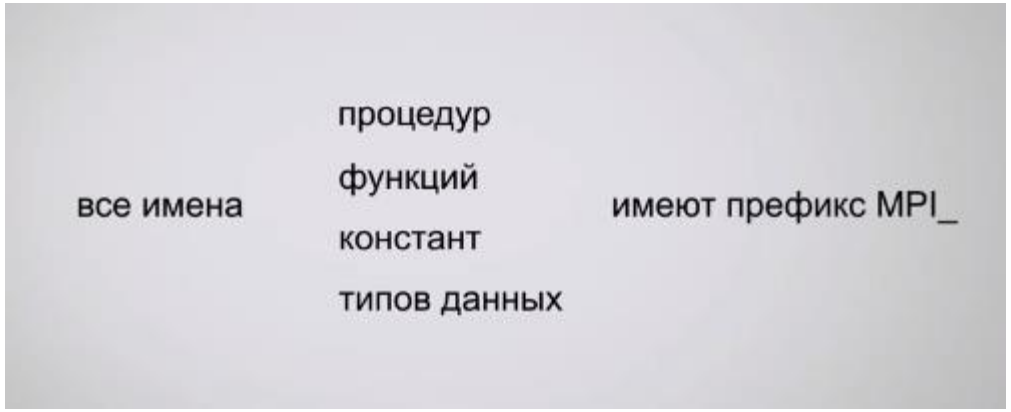
Определение:

Номер процесса именуется ***рангом*** процесса.

Модель вычислений



Для C/C++:



Все функции начинаются с большой буквы после префикса:

MPI_Comm_rank
MPI_Comm_size
MPI_Barrier

Все константы и типы данных записываются полностью заглавными буквами:

MPI_COMM_WORLD
MPI_INT

MPI: основные понятия и определения…

В основу MPI положены четыре основных понятия:

- Тип операции передачи сообщения
- Тип данных, пересылаемых в сообщении
- Понятие коммуникатора (*группы процессов*)
- Понятие виртуальной топологии

MPI: основные понятия и определения...

Операции передачи данных

Основу MPI составляют операции передачи сообщений.

- Среди предусмотренных в составе MPI функций различаются:
  - парные (*point-to-point*) операции между двумя процессами,
  - коллективные (*collective*) операции для одновременного взаимодействия нескольких процессов.

MPI: основные понятия и определения…

Понятие коммуникаторов

Определение:

Коммуникатор в MPI - специально создаваемый служебный объект, объединяющий в своем составе группу процессов и ряд дополнительных параметров (*контекст)*:

- парные операции передачи данных выполняются для процессов, принадлежащих одному и тому же коммуникатору,
- коллективные операции применяются одновременно для всех процессов одного коммуникатора.

Указание коммуникатора является обязательным для **всех** операций передачи данных в MPI.

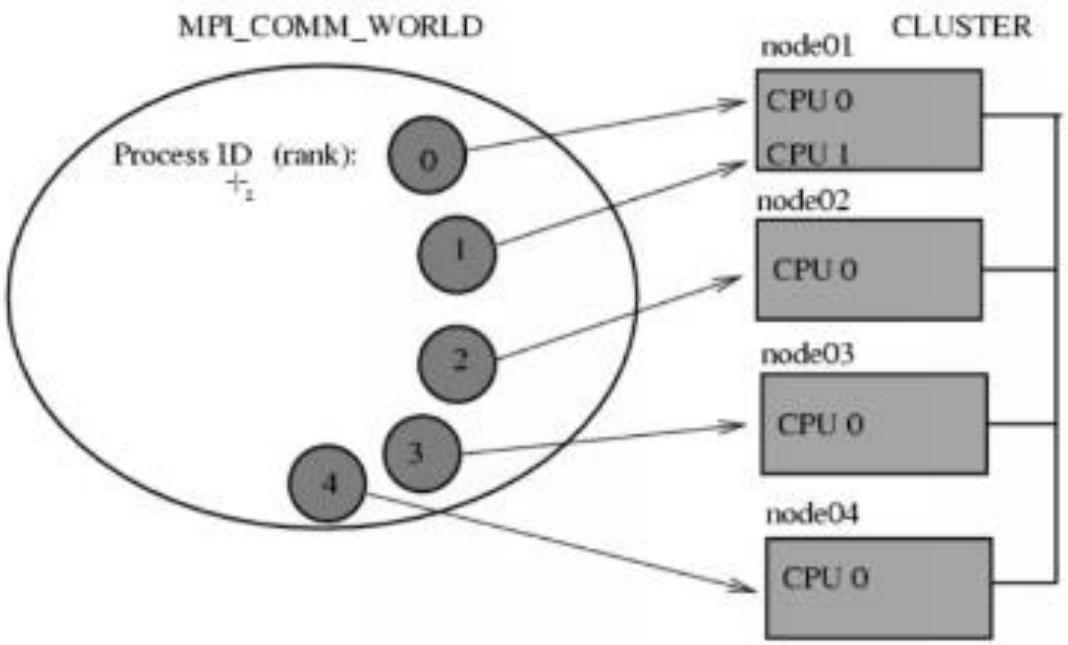
MPI: основные понятия и определения…

В ходе вычислений могут создаваться новые и удаляться существующие коммуникаторы.

Один и тот же процесс может принадлежать разным коммуникаторам.

Все имеющиеся в параллельной программе процессы входят в состав создаваемого *по умолчанию коммуникатора* с идентификатором **MPI_COMM_WORLD**.

Коммуникатор



Типы данных MPI

При выполнении операций передачи сообщений для определения передаваемых или получаемых данных в функциях MPI необходимо указывать тип пересылаемых данных.

MPI содержит большой набор *базовых типов данных*, во многом совпадающих с типами данных в языках C/C++ и Fortran.

В MPI можно создавать *новые производные типы данных* для более точного и краткого описания содержимого пересылаемых сообщений.

Типы данных для передачи и приёма сообщений

Базовые типы данных MPI для языка C/C++

MPI_Datatype	C Datatype
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Инициализация и завершение MPI программ

Первой вызываемой функцией MPI должна быть функция:

int **MPI_Init** (int *argc, char **argv);

(служит для *инициализации среды* выполнения MPI программы; параметрами функции являются количество аргументов в командной строке ОС и текст самой командной строки.)

Последней вызываемой функцией MPI обязательно должна являться функция:

int **MPI_Finalize** (void);

Инициализация и завершение MPI программ

Структура параллельной программы, разработанная с использованием MPI, должна иметь следующий вид:

```
#include "mpi.h"
int main ( int argc, char *argv[] ) {
  <программный код без использования MPI
функций>
  MPI_Init ( &agrc, &argv );
    <программный код с использованием MPI
функций >
  MPI_Finalize();
  <программный код без использования MPI
функций >
  return 0;
```

Первая программа

```
#include "mpi.h"
#include <iostream>
int main (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);
    std::cout << " Общее число ПЭ " <<  size <<
                 " мой номер " << rank << std::endl;
    MPI_Finalize ();
    return 0;
}

Общее число ПЭ  4,  мой номер  0
Общее число ПЭ  4,  мой номер  2
Общее число ПЭ  4,  мой номер  1
Общее число ПЭ  4,  мой номер  3
```

Определение количества и ранга процессов

Определение *количества процессов* в выполняемой параллельной программе осуществляется при помощи функции:

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

Для определения *ранга процесса* используется функция:

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

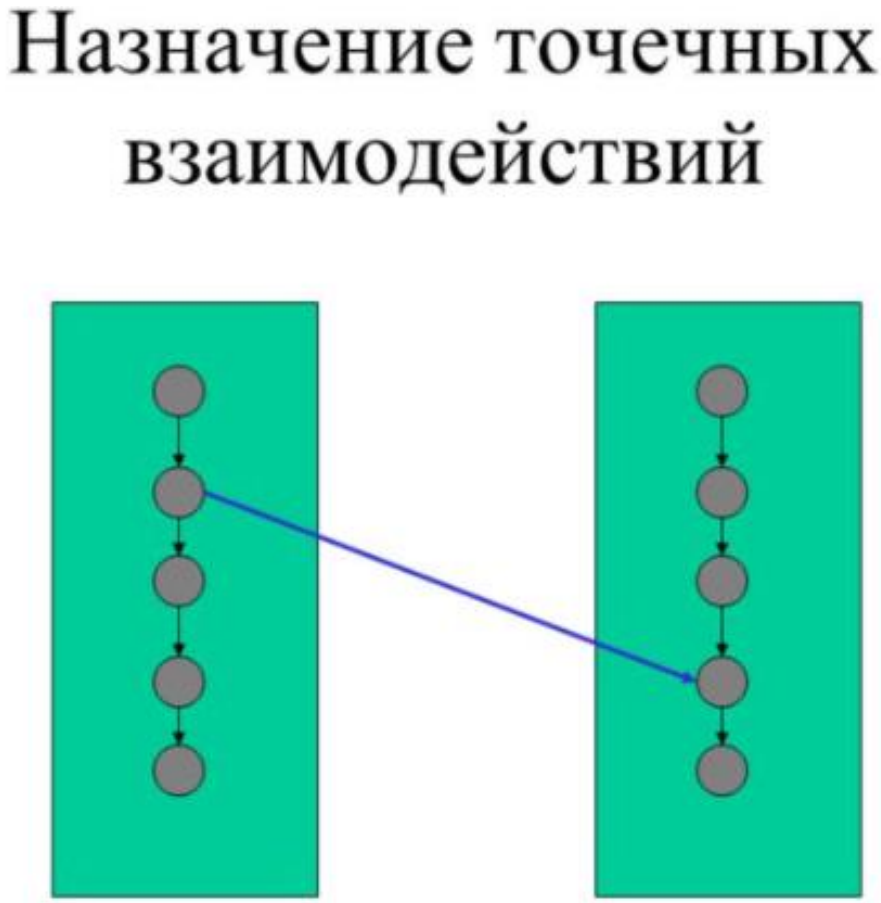
Определение количества и ранга процессов

Коммуникатор *MPI_COMM_WORLD* создается по умолчанию и представляет все процессы выполняемой параллельной программы;

Ранг, получаемый при помощи функции *MPI_Comm_rank*, является рангом процесса, выполнившего вызов этой функции, и, тем самым, переменная *ProcRank* будет принимать *различные значения в разных процессах*.

Работа функции MPI_Comm_rank()

Точечные взаимодействия.



Парная передача сообщений

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

Парная передача сообщений

Для *передачи сообщения* процесс-отправитель должен выполнить функцию:

int **MPI_Send**(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm);

где:

buf – адрес буфера памяти, в котором располагаются данные отправляемого сообщения,

count – количество элементов данных в сообщении,

type - тип элементов данных пересылаемого сообщения,

dest - ранг процесса, **которому** отправляется сообщение,

tag - значение-тег, используемое для идентификации сообщений,

comm - коммуникатор, в рамках которого выполняется передача данных.

Передача сообщений…

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

Прием сообщений

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

Прием сообщений

Для *приема сообщения* процесс-получатель должен выполнить функцию:

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

где

- **buf, count, type** – буфер памяти для приема сообщения
- **source** - ранг процесса, от которого должен быть выполнен прием сообщения,
- **tag** - тег сообщения, которое должно быть принято для процесса,
- **comm** - коммуникатор, в рамках которого выполняется передача данных,
- **status** – указатель на структуру данных с информацией о результате выполнения операции приема данных.

Прием сообщений…

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

Прием сообщений...

Параметр *status* позволяет определить ряд характеристик принятого сообщения:

status.MPI_SOURCE – ранг процесса-отправителя принятого сообщения,

status.MPI_TAG - тег принятого сообщения.

Функция

MPI_Get_count(MPI_Status *status, MPI_Datatype type, int *count)

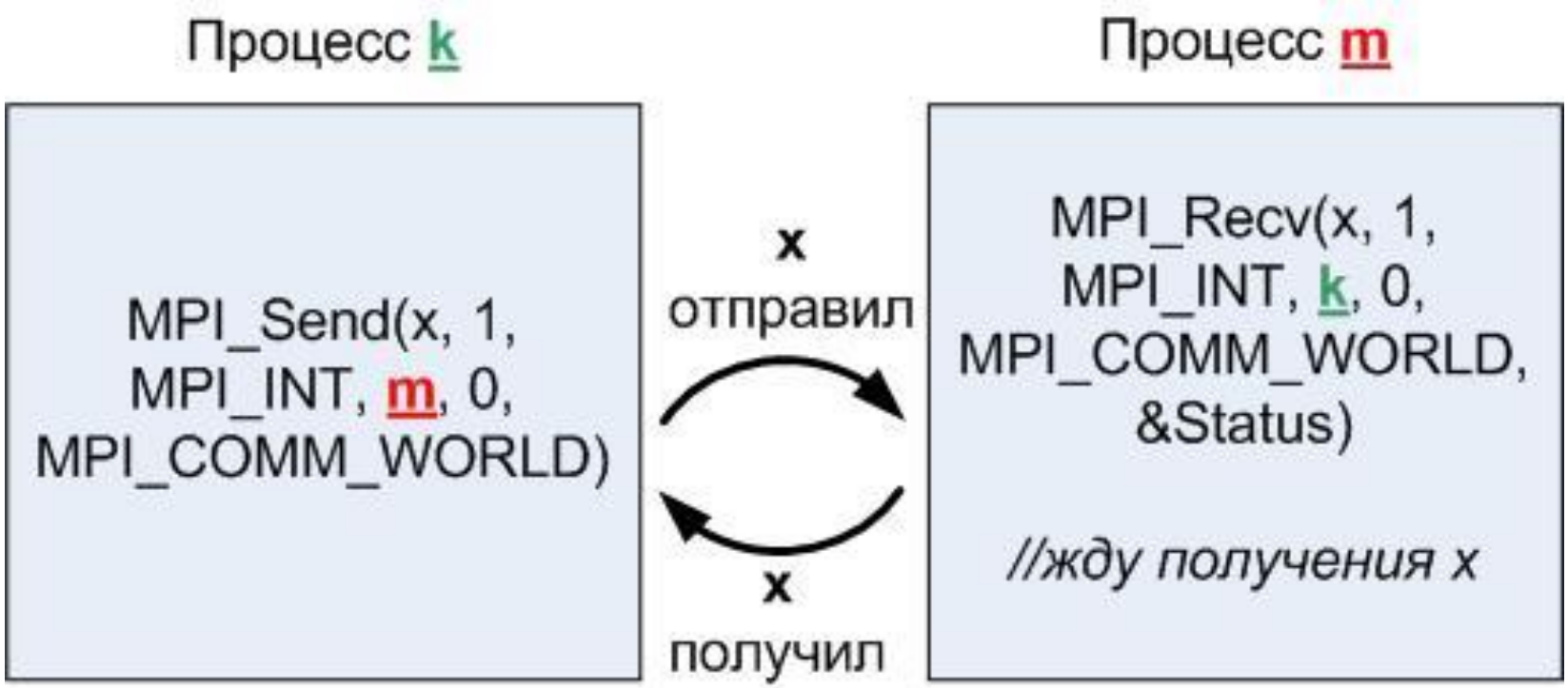
возвращает в переменной *count* количество элементов типа *type* в принятом сообщении.

Прием сообщений…

Функция *MPI_Recv* является *блокирующей* для процесса-получателя, т.е. его выполнение приостанавливается до завершения работы функции.

Таким образом, **если** по каким-то причинам **ожидаемое** для приема **сообщение будет отсутствовать**, **выполнение** параллельной **программы будет блокировано**.

Парные функции приема-передачи сообщений



Первая параллельная программа с использованием MPI…

- Каждый процесс определяет свой ранг, после чего действия в программе разделяются (разные процессы выполняют различные действия),
- Все процессы, кроме процесса с рангом 0, передают значение своего ранга нулевому процессу,
- Процесс с рангом 0 сначала печатает значение своего ранга, а далее последовательно принимает сообщения с рангами процессов и также печатает их значения.

Первая параллельная программа с использованием MPI…

Порядок приема сообщений заранее не определен и зависит от условий выполнения параллельной программы (более того, этот порядок может изменяться от запуска к запуску).

Если это не приводит к потере эффективности, следует обеспечивать однозначность расчетов и при использовании параллельных вычислений:

Обмен сообщениями между двумя процессами

Нулевой процесс посылает сообщение процессу с номером 1 и ждет от него ответа.

Программа запущена на 3-х процессах, выполняют пересылки 0 и 1.

```
process 2 a = 0 b = 0
process 0 a = 2 b = 1
process 1 a = 2 b = 1
```

```
int main(int argc, char **argv) {
  int rank;  float a, b;        MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  a = 0.0;              b = 0.0;
  if ( rank  ==  0 ) {
    b = 1.0;
    MPI_Send(&b, 1, MPI_INT, 1, 5, MPI_COMM_WORLD);
    MPI_Recv(&a, 1, MPI_INT, 1, 5, MPI_COMM_WORLD, &status);
  }
  if ( rank  ==  1 ) {
    a = 2.0;
    MPI_Recv(&b, 1, MPI_FLOAT, 0, 5, MPI_COMM_WORLD, &status);
    MPI_Send(&a, 1, MPI_FLOAT, 0, 5, MPI_COMM_WORLD);
  }
  cout << " process " << rank <<  " a = " << a << " b = " << b << endl;
  MPI_Finalize();
}
```

13

Вычисление числа Пи.

```
#include <stdio.h>
#include "mpi.h"
#include <sys\timeb.h>
double realtime(void);
double compute_interval (int myrank, int ntasks, long intervals)
{
 double width, x, localsum;
 long j;
 width = 1.0 / intervals;/* width of single stripe */
 localsum = 0.0;
 for (j = myrank; j < intervals; j += ntasks)
 {
       x = (j + 0.5) * width;
       localsum += 4 / (1 + x * x);
 }
 return (localsum * width); /* size of area */
}
```

```
void main(int argc, char ** argv)
{
 long intervals;
 int myrank, ranksize;
 double pi, di , send[2],recv[2];
 int i;  MPI_Status status;
 double t1,t2;    double t3, t4, t5, t6,t7;
 t1=realtime();
 MPI_Init (&argc, &argv);       /* initialize MPI system */
 t2=realtime();
 MPI_Comm_rank (MPI_COMM_WORLD, &myrank);    /* my place in MPI system
*/
 MPI_Comm_size (MPI_COMM_WORLD, &ranksize);  /* size of MPI system */
 if (myrank == 0)              /* I am the master */
 {
     printf ("Calculation of PI by numerical Integration\n");
     printf ("Number of intervals: ");
     scanf ("%ld", &intervals);
 }
```

```
MPI_Barrier (MPI_COMM_WORLD);  /* make sure all MPI tasks are running */
 if (myrank == 0)              /* I am the master */
 { /* distribute parameter */
printf ("Master: Sending # of intervals to MPI-Processes \n");
t3 = MPI_Wtime();
for (i = 1; i < ranksize; i++)
{
MPI_Send (&intervals, 1, MPI_LONG, i, 98, MPI_COMM_WORLD);
}
 }
 else{/* I am a slave */
 /* receive parameters */
  MPI_Recv (&intervals, 1, MPI_LONG, 0, 98, MPI_COMM_WORLD, &status);
 }
 /* compute my portion of interval */
 t4 = MPI_Wtime();
 pi = compute_interval (myrank, ranksize, intervals);
 t5 = MPI_Wtime();
 MPI_Barrier (MPI_COMM_WORLD);
```

```
t6 = MPI_Wtime();

 if (myrank == 0)/* I am the master */
 /* collect results, add up, and print results */
 {
for (i = 1; i < ranksize; i++)
{
MPI_Recv (&di, 1, MPI_DOUBLE, i, 99, MPI_COMM_WORLD, &status);
pi += di;
}
t7 = MPI_Wtime();
printf ("Master: Has collected sum from MPI-Processes \n");
printf ("\nPi estimation: %.20lf\n", pi);
printf ("%ld tasks used - Execution time: %.3lf sec\n",ranksize, t7 -t3);
printf("\nStatistics:\n");
printf("Master: startup: %.0lf msec\n",t2-t1);
printf("Master: time to send # of intervals:%.3lf sec\n",t4-t3);
printf("Master: waiting time for sincro after calculation:%.2lf sec\n",t6-t5);
printf("Master: time to collect: %.3lf sec\n",t7-t6);
printf("Master: calculation time:%.3lf sec\n",t5-t4);
MPI_Barrier (MPI_COMM_WORLD);
```

```
/* collect there calculation time */
for (i = 1; i < ranksize; i++){
MPI_Recv (recv, 2, MPI_DOUBLE, i, 100, MPI_COMM_WORLD, &status);
printf("process %d: calculation time: %.3lf sec\twaiting time for sincro.: %.3lf sec\n",i,recv[0],recv[1]);
}
 }
 else{/* I am a slave */
 /* send my result back to master */
  MPI_Send (&pi, 1, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD);

  MPI_Barrier (MPI_COMM_WORLD);
  send[0]=t5-t4;
  send[1]=t6-t5;
  MPI_Send (send, 2, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD);
 }
 MPI_Finalize ();
}
double realtime()     /* returns time in seconds */
{
struct _timeb tp;
_ftime(&tp);
return((double)(tp.time)*1000+(double)(tp.millitm));
}
```

Итоги

Мы рассмотрели:

Основные определения и понятия MPI, основные функции MPI и их применение.