Суперкомпьютерное моделирование и технологии

Лекция

Технология параллельного программирования MPI

Часть 1.

.

Попова Нина Николаевна

доцент кафедры СКИ

popova@cs.msu.su

7 октября 2022 г.

Архитектура параллельных вычислительных систем

2 основных класса параллельных ВС:

- **системы с общей памятью** (многоядерные процессоры, специальные SMP-процессоры)

- **системы с распределенной памятью**

  вычислительные узлы =

  многоядерные процессоры +

  графические ускорители

Модели параллельных программ

- **Системы с общей памятью**
  - Программирование основано на потоках
  - Программа строится на базе последовательной программы
  - Возможно автоматическое распараллеливание компилятором с использованием соответствующего ключа компилятора
  - Директивы компиляторов (**OpenMP**, OpenACC,…)

- **Системы с распределенной памятью**
  - Программа состоит из параллельных процессов
  - Явное задание коммуникаций между процессами – обмен сообщениями **“Message Passing”**

  Реализация - Message Passing библиотек:
    - MPI (“Message Passing Interface”)
    - PVM (“Parallel Virtual Machine”)
    - Shmem,

MPI

- MPI реализует модель передачи сообщений:

  Вычисления – множество процессов, взаимодействующих друг с другом посредством передачи сообщений

- MPI – **спецификация**, не язык

- Реализация спецификаций MPI реализуется в виде библиотек функций , вызываемых из языков программирования высокого уровня: C, Fortran

OpenMP

- OpenMP : **Open** specifications for **M**ulti **P**rocessing

  – многопоточный интерфейс, специально разработанный для поддержки параллельных вычислений

  - не предназначен для систем с распределенной памятью, не поддерживает передачу сообщений

  - рекомендуется для достижения максимального параллелизма на системах с общей памятью

Пример параллельной программы программы (C, OpenMP)

Сумма элементов массива

```
#include <stdio.h>
#define N 100000
int main()
{ double sum;
  double a[N];
  int   i, n =N;
  for (i=0; i<n; i++){
    a[i] = i*0.5;  }
 sum =0;
#pragma omp parallel for reduction (+:sum)
 for (i=0; i<n; i++)
  sum = sum+a[i];
printf (“Sum=%f\n”, sum);
}
```

Пример параллельной программы программы (C, MPI)

```
#include <stdio.h>
#include <mpi.h>
#define N 100000
int main(int argc, char *argv[])
{ double sum, local_sum;
  double *a;
  int  i, n =N;
  int size, myrank;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,
      &rank);
MPI_Comm_size(MPI_COMM_WORLD,
      &size);
n= n/ size;
a= (double *) malloc(n);
```

```
for (i=0;  i<n; i++)
{
    a[i] = (rank*n +i)*0.5;
 }
 local_sum =0;
for (i=0; i<n; i++)
   local_sum += a[i];
MPI_Reduce(&local_sum, &sum, 1,
 MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
If ( !rank)
    printf ("Sum =%f\n", sum);
MPI_Finalize();
return 0;
}
```

Гибрид: MPI+OpenMP

```
#include <stdio.h>
#include <mpi.h>
#define N 1024
int main(int argc, char *argv[])
{ double sum, local_sum;
  double *a;
  int  i, n =N;
  int size, myrank;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,
      &rank);
MPI_Comm_size(MPI_COMM_WORLD,
      &size);
n= n/ size;
a= (double *) malloc(n);
```

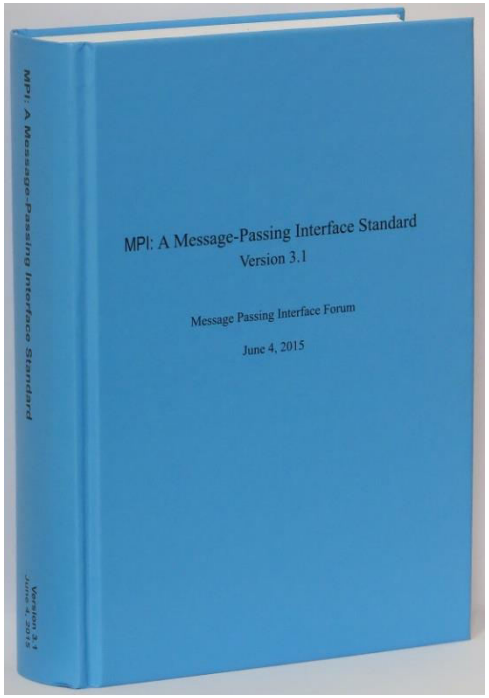
```
  for (i=0;  i<n; i++)
{
    a[i] = (rank*n+i)*0.5;
 }
 local_sum =0;
#pragma omp parallel for reduction
(+:sum)
for (i=0; i<n; i++)
  local_sum += a[i];
MPI_Reduce(&local_ sum,& sum, 1,
 MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
If ( !rank)
printf (“Sum =%f\n”, sum);
MPI_Finalize();
return 0;
}
```

MPI стандарт для построения параллельных программ для вычислительных систем с распределенной памятью

MPI – стандарт (формальная спецификация)

- MPI 1.1 Standard разрабатывался 92-94
- MPI 2.0 - 95-97
- MPI 2.1 - 2008
- MPI 3.0 – 2012
- MPI 3.1 - 2015
- MPI 4.0 - 2021
- Стандарты
  - **http://www.mcs.anl.gov/mpi**
  - **http://www.mpi-forum.org/docs/docs.html**
  - https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

  Описание функций
  - **http://www-unix.mcs.anl.gov/mpi/www/**



868 стр.

Реализации MPI - библиотеки

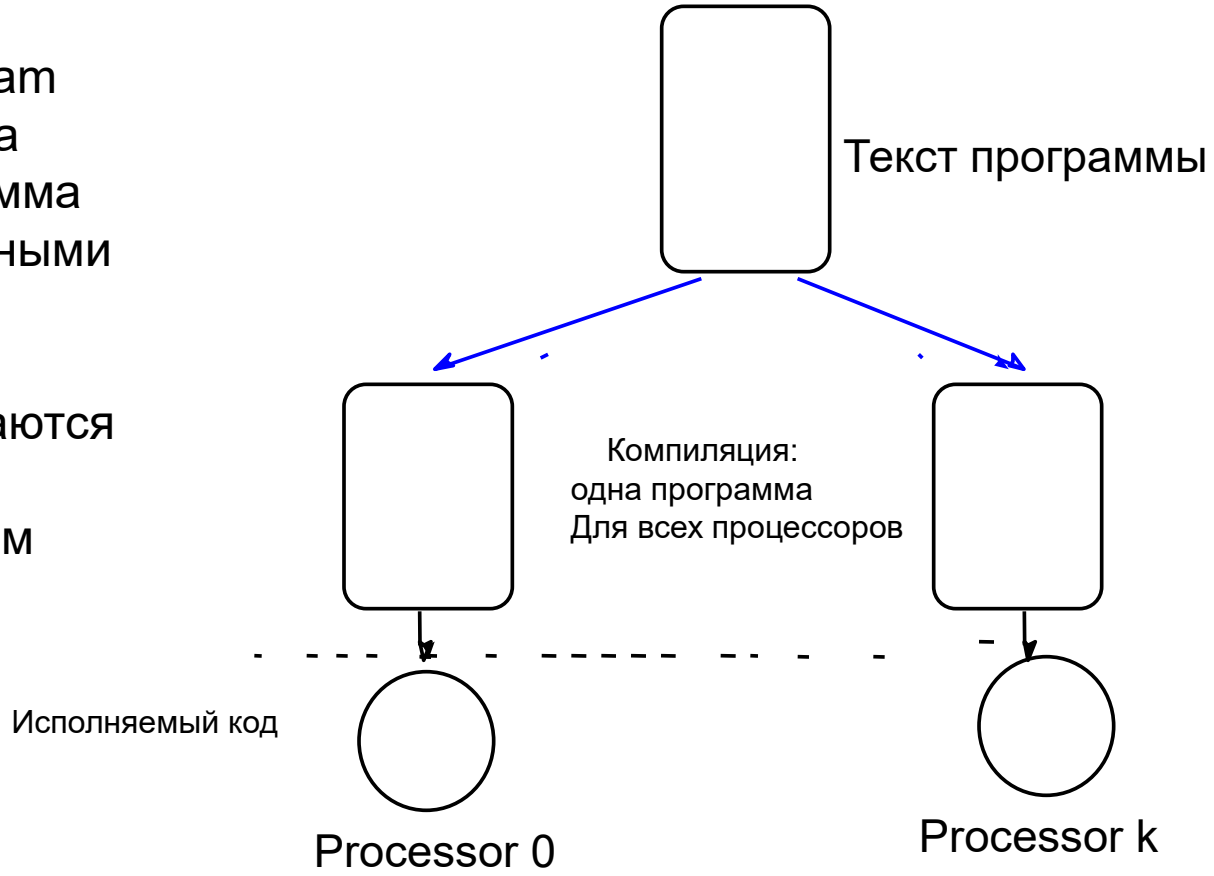
- MPICH (www.mpich.org , Argonne)
- LAM/MPI (в настоящее время не поддерживается)
- OpenMPI (www.open-mpi.org
  Los Alamos, Unis of Tennessee, Indiana & Stuttgart)
- Mvapich
- Коммерческие реализации Intel,IBM и др.

Модель MPI

- Параллельная программа состоит из процессов, процессы могут быть многопоточными.
- MPI реализует передачу сообщений между процессами.
- Межпроцессное взаимодействие предполагает:
  - синхронизацию
  - перемещение данных из адресного пространства одного процесса в адресное пространство другого процесса.

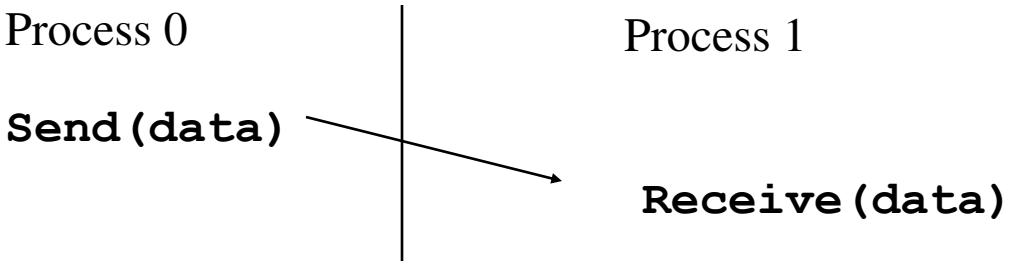
Модель MPI-программ

- **SPMD** – Single Program Multiple Data
- Одна и та же программа выполняется различными процессорами
- Управляющими операторами выбираются различные части программы на каждом процессоре.



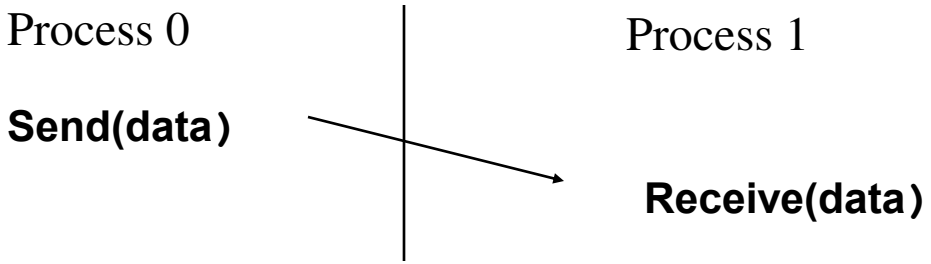
Основы передачи данных в MPI

- Данные посылаются одним процессом и принимаются другим.
- Передача и синхронизация совмещены.



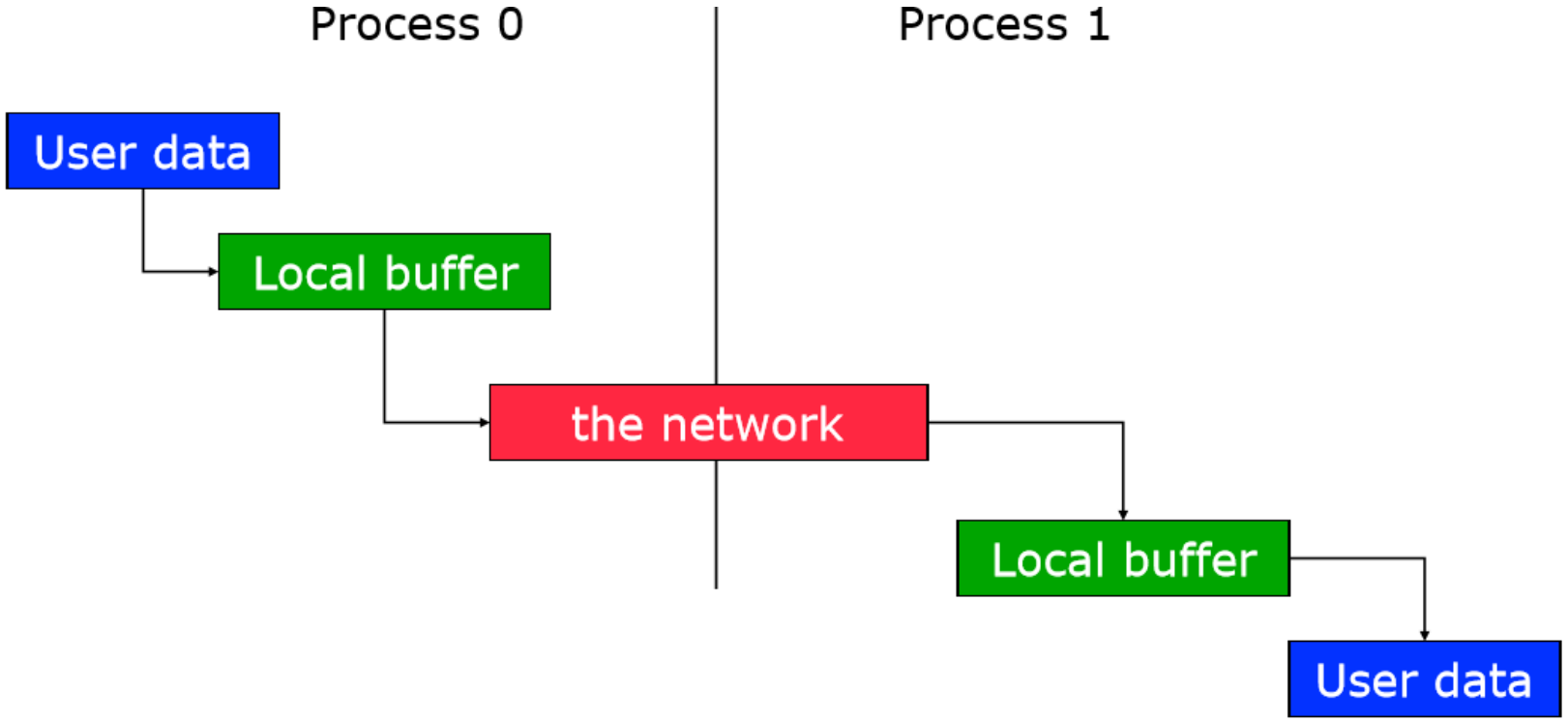
Основы передачи данных в MPI

- Необходимы уточнения процесса передачи



- Требуется уточнить:
  - Как должны быть описаны данные ?
  - Как должны идентифицироваться процессы?
  - Как получатель получит информацию о сообщении?
  - Что значить завершение передачи?

Возможная схема выполнения операций передачи сообщений



6 основных функций MPI

- **Как стартовать/завершить параллельное выполнение**
  - MPI_Init
  - MPI_Finalize
- **Кто я (и другие процессы), сколько нас**
  - MPI_Comm_rank
  - MPI_Comm_size
- **Как передать сообщение коллеге (другому процессу)**
  - MPI_Send
  - MPI_Recv

Основные понятия MPI

- Процессы объединяются в ***группы***.
- Группе приписывается ряд свойств (как связаны друг с другом и некоторые другие). Получаем ***коммуникаторы***
- Процесс идентифицируется своим номером в группе, привязанной к конкретному коммуникатору.
- При запуске параллельной программы создается специальный коммуникатор с именем ***MPI_COMM_WORLD***

Понятие коммуникатора MPI

- **Все** обращения к MPI функциям содержат коммуникатор, как параметр.

- Наиболее часто используемый коммуникатор **`MPI_COMM_WORLD`**
  - определяется при вызове **`MPI_Init`**
  - содержит ВСЕ процессы программы

Типы данных MPI

- Данные в сообщении описываются тройкой: (***address, count, datatype***)
- ***datatype*** (типы данных MPI)

Signed
- MPI_CHAR
- MPI_SHORT
- MPI_INT
- MPI_LONG

Unsigned
- MPI_UNSIGNED_CHAR
- MPI_UNSIGNED_SHORT
- MPI_UNSIGNED
- MPI_UNSIGNED_LONG

- MPI_FLOAT
- MPI_DOUBLE
- MPI_LONG_DOUBLE

MPI datatype	C datatype
MPI_CHAR	char (treated as printable character)
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG (as a synonym)	signed long long int
MPI_SIGNED_CHAR	signed char (treated as integral value)
MPI_UNSIGNED_CHAR	unsigned char (treated as integral value)
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t (defined in <stddef.h>) (treated as printable character)
MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_BYTE	
MPI_PACKED	

Тэг собщения

- Сообщение сопровождается определяемым пользователем признаком – целым числом – ***тэгом*** для идентификации принимаемого сообщения.
- Теги сообщений у отправителя и получателя должны быть согласованы. Можно указать в качестве значения тэга константу **`MPI_ANY_TAG`**.
- Некоторые не-MPI системы передачи сообщений называют тэг типом сообщения. MPI вводит понятие тэга, чтобы не путать это понятие с типом данных MPI.

Формат MPI-функций

C :

```
int MPI_Xxxxx(parameter,...);
```

C++ :

```
int MPI::Xxxxx(parameter,...);
```

Функции (за некоторым исключением) возвращают код ошибки.
По умолчанию программа завершается при возникновении ошибки.

Шаблон MPI-программы

```
#include < stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char **argv)
 {
// непараллельная часть программы
MPI_Init(&argc, &argv);
 /* анализ аргументов */
 /*  программа */
  MPI_Finalize();
  exit (0);
 }
```

Начало параллельного выполнения

Старт параллельного выполнения . Этот код выполняется всеми процессами

Завершение параллельного выполнения

Hello, MPI world!

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

Коммуникатор MPI_COMM_WORLD

ID процесса

Число процессов

Трансляция MPI-программ

- Трансляция

  mpixlc –o <имя_программы> <имя>.c <опции>

  Например:

  mpixlc –o hw helloworld.c
- Запуск в интерактивном режиме

  mpirun –np 128 hw
- Постанока в очередь (Blue Gene/P)

  mpisubmit.bg –n 128 ./hw

Компиляция MPI-программ на Polus

- *module avail*
- *module load SpectrumMPI/10.1.0*
- Компиляция

  mpixlc –o hw helloworld.c

- Постановка на выполнение

  mpisubmit.pl [параметры скрипта] исполняемый_файл [-- параметры исполняемого файла]

  mpisubmit.pl –n 32 hw

Очереди Polus

- Проверка состояния очереди

  bjobs

или

  bjobs –u all

- Просмотр информации о задаче

  bjobs –l <job_ID>

- Какие очереди есть

  bqueues

- Удалить задачу из очереди

  bkill job_ID

Функции определения среды

```
int MPI_Init(int *argc, char ***argv)
```
должна первым вызовом, вызывается только один раз

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```
число процессов в коммуникаторе

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```
номер процесса в коммуникаторе (нумерация с 0)

```
int MPI_Finalize()
```
завершает работу процесса

```
int MPI_Abort (MPI_Comm_size(MPI_Comm comm,
int*errorcode)
```
завершает работу программы

Инициализация MPI

- **MPI_Init** должна первым вызовом, вызывается только один раз

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

- Проверка была ли выполнена инициализация:

```
int MPI_Initialized( int *flag )
```

flag = true, если MPI уже инициализирована.

Завершение MPI-процессов

- Никаких вызовов MPI функций после

  int MPI_Finalize()

  int MPI_Abort(MPI_Comm comm, int errorcode)

  errorcode - код ошибки для возврата в среду исполнения.

Пользователь обязан гарантировать, что все незаконченные обмены будут завершены прежде, чем будет вызвана MPI_FINALIZE.

Если какой-либо из процессов не выполняет MPI_Finalize, программа зависает.

Количество процессов в коммуникаторе

- Размер коммуникатора

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

Результат – число процессов в коммуникаторе

MPI_Comm_rank
номер процесса (process rank)

- Process ID в коммуникаторе
  - Начинается с 0 до (*n*-1), где *n* – число процессов
- Используется для определения номера процесса в коммуникаторе

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

Результат – номер процесса

Взаимодействие «точка-точка»

- Самая простая форма обмена сообщением
- Один процесс посылает сообщения другому
- Несколько вариантов реализации того, как пересылка и выполнение программы совмещаются

MPI_Send

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

buf		адрес буфера
count	–	число пересылаемых элементов
Datatype	–	MPI datatype
dest	-	rank процесса-получателя
tag	-	определяемый пользователем параметр,
comm	–	MPI-коммуникатор

Пример:

```
MPI_Send(data,500,MPI_FLOAT,6,33,MPI_COMM_WORLD)
```

MPI_Recv

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

buf адрес буфера
count – число пересылаемых элементов
Datatype – MPI datatype
source - rank процесса-отправителя
tag - определяемый пользователем параметр,
comm – MPI-коммуникатор,
status - статус

Пример:

```
MPI_Recv(data,500,MPI_FLOAT,6,33,MPI_COMM_WORLD,&stat)
```

Завершение

- “Завершение” передачи означает, что буфер в памяти, занятый для передачи, может быть безопасно использован для доступа, т.е.
  - Send: переменная, задействованная в передаче сообщения, может быть доступна для дальнейшей работы
  - Receive: переменная, получающая значение в результате передачи, может быть использована

Информация о завершившемся приеме сообщения

- Возвращается функцией **`MPI_Recv`** через параметр **`status`**
- Содержит:
  - Source: ***`status.MPI_SOURCE`***
  - Tag: ***`status.MPI_TAG`***
  - Count: ***`MPI_Get_count`***
- Если status не нужен -> можно задать значение **`MPI_STATUS_IGNORE`**

Полученное сообщение

- Может быть меньшего размера, чем указано в функции MPI_Recv, но НЕ БОЛЬШЕ!
- `count` – число реально полученных элементов

Функция для определения числа полученных элементов:

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

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
   int rank, size; int x; MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   x=rank;
    if(!rank) {
     MPI_Send (&x,1, MPI_INT, 1,0,MPI_COMM_WORLD);
      printf(" I am %d. I sent my rank to %d\n“,x,1);
    } else {
    MPI_Recv (&x,1, MPI_INT,
      0,0,MPI_COMM_WORLD,&status);
    printf(" I am %d. I received   %d  from \n“,rank,x); }
   MPI_Finalize();
   return 0;
 }
```

0-ой процесс посылает x 1-ому процессу

1-ый процесс принимает целое число в переменную x

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
   int rank, size; int x; MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   x=rank;
    if(!rank) {
     MPI_Send (&x,1, MPI_INT, 1,0,MPI_COMM_WORLD);
     printf(" I am %d. I sent my rank to %d\n“,x,1);
    } else {
    MPI_Recv (&x,1, MPI_INT,
     0,0,MPI_COMM_WORLD,&status);
     printf(" I am %d. I received   %d  from \n“,rank,x); }
  MPI_Finalize();
   return 0;
} /*Программа будет работать правильно
    только для запуска на 2-ух процессах.
    ПОЧЕМУ? */
```

0-ой процесс посылает x 1-ому процессу

1-ый процесс принимает целое число в переменную x

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
   int rank, size; int x; MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
    x=rank;
    if(!rank) {
      for (int i=1; i<size; i++)
        {MPI_Send (&x,1, MPI_INT, i,0,MPI_COMM_WORLD);
         printf(" I am %d. I sent my rank to %d\n“,x,i); }
     else {
        MPI_Recv (&x,1, MPI_INT, 0, 0,  MPI_COMM_WORLD,
     &status);
         printf(" I am %d. I received %d from process\n“,rank,0);
     }
   MPI_Finalize();   return 0; }
```

0-ой процесс посылает x ВСЕМ остальным процессам

1-ый процесс принимает целое число в переменную x

Wildcarding (джокеры)

- Получатель может использовать специальный «джокер» для получения сообщения от **ЛЮБОГО** процесса
  `MPI_ANY_SOURCE`
- Для получения сообщения с ЛЮБЫМ тэгом
  `MPI_ANY_TAG`
- Реальные номер процесса-отправителя и тэг возвращаются через параметр ***`status`***

Нулевые процессы

- В качестве отправителя или получателя, вместо номера может быть использовано специальное значение MPI_PROC_NULL.
- Обмен с процессом, который имеет значение MPI_PROC_NULL, не дает результата.
- Передача в процесс MPI_PROC_NULL успешна и заканчивается сразу, как только возможно.
- Прием от процесса MPI_PROC_NULL успешен и заканчивается сразу, как только возможно без измененения буфера приема.
-
- При выполнении приема из source = MPI_PROC_NULL в статус возвращает source = MPI_PROC_NULL, tag = MPI_ANY_TAG и count = 0

MPI_Probe

int MPI_Probe(int *source*, int *tag*, MPI_Comm *comm*, MPI_Status* *status*)

Проверка статуса операции приема сообщения.

Параметры аналогичны функции MPI_Recv

Условия успешного взаимодействия «точка-точка»

- Отправитель должен указать правильный rank получателя
- Получатель должен указать верный rank отправителя
- Одинаковый коммуникатор
- Тэги должны соответствовать друг другу
- Буфер у процесса-получателя должен быть достаточного объема

Замер времени MPI_Wtime

- Время замеряется в секундах

```
double MPI_Wtime(void)
```

Пример.

```
double start, finish, time ;
start=-MPI_Wtime;
MPI_Send(…);
finish = MPI_Wtime();
time= start+finish;
```

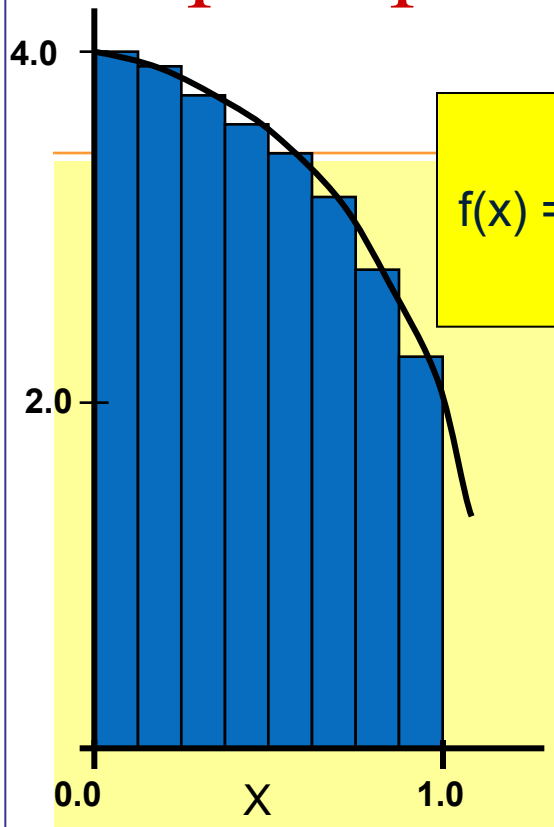
Сумма элементов вектора с использованием MPI_Send и MPI_Recv

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define N_LOCAL 1024
int main(int argc, char *argv[])
{ double sum_local, sum_global, start, finish;
  double a[N_LOCAL];
  int  i, n =N_LOCAL;
  int size, myrank;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Сумма элементов вектора с использованием MPI_Send и MPI_Recv

```
If (!myrank)  /* process-root */
{ start = MPI_Wtime(); sum_global=0;
  for (i=1; i<size; i=+) {
  MPI_Recv( &sum_local, 1,  MPI_DOUBLE, MPI_ANY_SOURCE,
  MPI_ANY_TAG, MPI_COMM_WORLD, &status);
  sum_global += sum_local; }
  printf (“ Sum of vector elements = %f \n Time =%f\n”, sum_global,
MPI_Wtime() - start);
}
 else { /* processes: 1 .. size-1 */
//Инициализация элементов вектора a
sum_local =0;
for(i=0; i<n; i++) sum_local+= a[i];
 MPI_Send (&sum_local, 1 ,  MPI_DOUBLE, 0,0, MPI_COMM_WORLD);
}
MPI_Finalize();
 exit (0); }
```

Пример: численное интегрирование



$$f(x) = \frac{4.0}{(1+x^2)}$$

```
static long num_steps=100000;
 double step, pi;

void main()
{  int i;
   double x, sum = 0.0;

   step = 1.0/(double) num_steps;
   for (i=0; i< num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0 + x*x);
   }
   pi = step * sum;
   printf("Pi = %f\n",pi);
}
```

Вычисление числа π с использованием MPI

```
#include "mpi.h"
#include <stdio.h>
int main (int argc, char *argv[])
{
    int n =100000, myid, numprocs, i;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    h   = 1.0 / (double) n;
    sum = 0.0;
```

Вычисление числа π с использованием MPI

```
for (i = myid + 1; i <= n; i += numprocs)
  {
    x = h * ((double)i - 0.5);
    sum += (4.0 / (1.0 + x*x));
  }
  mypi = h * sum;
```

Вычисление числа π с использованием MPI

```
if (!myid){
   pi= mypi;
   for (i= 1; i<numproc; i++){
    MPI_Recv (&mypi,1, MPI_DOUBLE, MPI_ANY_SOURCE,
     MPI_ANY_TAG, MPI_COMM_WORLD,MPI_STATUS_IGNORE );
     pi+=mypi; }
    printf ( "PI is approximately %.16f ", pi);
     }
else
  MPI_Send(&mypi, 1,MPI_DOUBLE,0, 0, MPI_COMM_WORLD);
MPI_Finalize();
    return 0;
}
```

Литература

- *Антонов А. С.* Технологии параллельного программирования MPI и OpenMP: Учеб. пособие. Предисл.: В.А.Садовничий. — Издательство Московского университета М.:, 2012. — С. 344.
- Интернет ресурсы:
- www.parallel.ru, intuit.ru, www.mpi-forum.org, http://www.mcs.anl.gov/research/projects/mpi/www/www3