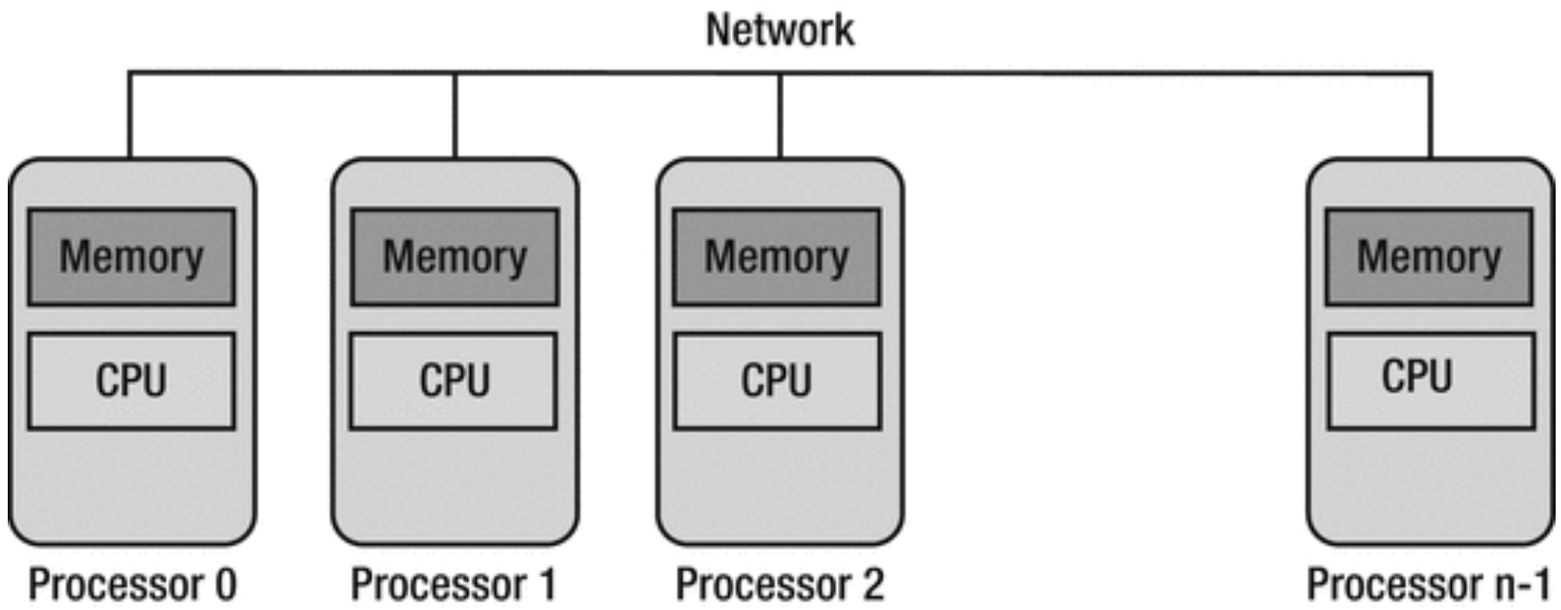
MPI (Message Passing Interface).



MPI - стандартизованный механизм для построения параллельных программ в модели обмена сообщениями, разработанная группой MPI Forum.

Версии стандарта MPI.

MPI 1 - 1994 год.

MPI 1.1 - 12 июля 1995 года.

- передача и получение сообщений между отдельными процессами;
- коллективные взаимодействия процессов;
- взаимодействия в группах процессов;
- реализация топологий процессов;

MPI 2.0 - 18 июля 1997 года.

- динамическое порождение процессов и управление процессами;
- односторонние коммуникации;
- параллельный ввод и вывод;
- расширенные коллективные операции.

MPI 2.1 - 1 сентябрь 2008 года.

MPI 2.2 - 4 сентябрь 2009 года.

MPI 3.0 - 21 сентябрь 2012 года.

MPI 3.1 - 4 июня 2015 года

Реализации MPI.

- MPICH2 (Open source, Argone NL)
- MVAPICH2
- IBM MPI
- Cray MPI
- Intel MPI
- HP MPI
- SiCortex MPI
- Open MPI (Open source, BSD License)
- Oracle MPI
- MPJ Express - MPI реализация для Java
- WMPI - реализация для Windows

Single Program, Multiple Data

```
#include <stdio.h>
#include "mpi.h"
#define MAX 100

int main(int argc, char **argv)
{
   int rank, size, n, i, ibeg, iend;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   n=(MAX-1)/size+1;
   ibeg=rank*n+1;
   iend=(rank+1)*n;
   for(i=ibeg; i<=((iend>MAX)?MAX:iend); i++)
      printf ("process %d, %d^2=%d\n", rank, i,
i*i);
   MPI_Finalize();
}
```

Стандартные коммуникаторы.

`MPI_COMM_WORLD` - коммуникатор, содержащий все процессы приложения

`MPI_COMM_SELF` - коммуникатор, содержащий только текущий процесс

`MPI_COMM_NULL` - коммуникатор, не содержащий процессов

Возвращаемые значения.

`MPI_SUCCESS` — Ошибки нет
`MPI_ERR_BUFFER` — Неправильный указатель буфера
`MPI_ERR_COUNT` — Неверное количество аргумента
`MPI_ERR_TYPE` — Неправильный тип аргумента
`MPI_ERR_TAG` — Неправильный тэг аргумента
`MPI_ERR_COMM` — Неправильный коммуникатор
`MPI_ERR_RANK` — Неправильный номер
`MPI_ERR_REQUEST` — Неверный запрос (дескриптор)
`MPI_ERR_ROOT` — Неверный корневой идентификатор

Общие процедуры MPI.

```
int MPI_Init(int *argc, char ***argv);
int MPI_Finalize(void);
int MPI_Comm_size(MPI_Comm comm, int *size);
int MPI_Comm_rank(MPI_Comm comm, int *rank);
double MPI_Wtime(void);
double MPI_Wtick(void);
int MPI_Get_processor_name(char *name, int
*len);
```

Функция инициализации `MPI_Init` позволяет активировать выполнение параллельной части программы. Вызов этой функции неявно вызывает барьерную синхронизацию, которая дожидается запуска всех нод, которые есть в данном кластере. В функцию `MPI_Init` передаются указатели на аргументы командной строки `argc` и `argv`, из которых системой могут извлекаться и передаваться в параллельные процессы параметры запуска программы. Если параметры командной стр не требуется, то могут передаваться значения `NULL`.

`int MPI_Finalize(void)` — завершение параллельной части приложения. Все последующие обращения к большинству процедур `MPI`, в том числе к `MPI_Init`, запрещены. К моменту вызова `MPI_Finalize` каждым процессом программы все действия, требующие его участия в обмене сообщениями, должны быть завершены.

Все функции `MPI` могут быть вызваны лишь после инициализации с помощью `MPI_Init` и до завершения с помощью `MPI_Finalize`. Стандарт предусматривает возможность проверки нахождения кода `MPI` в валидном положении с помощью проверочных функций `int MPI_Initialized(int *flag)` и `int MPI_Finalized(int *flag)`. В аргументе `flag` возвращает `1`, если функция вызвана после процедуры `MPI_Init/MPI_Finalize`, и `0` - в противном случае. Эти процедуры можно вызвать до `MPI_Init` и после `MPI_Finalize`.

`int MPI_Comm_size(MPI_Comm comm, int *size)` — в аргументе `size` возвращает число параллельных процессов в коммуникаторе comm.

`int MPI_Comm_rank(MPI_Comm comm, int *rank)` - в аргументе `rank` возвращает номер процесса в коммуникаторе comm в диапазоне от `0` до `size - 1`.

`double MPI_Wtime(void)` — возвращает для каждого вызвавшего процесса астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом. Момент времени, используемый в качестве точки отсчёта, не будет изменён за время существования процесса.

`double MPI_Wtick(void)` — возвращает разрешение таймера в секундах.

`int MPI_Get_processor_name(char *name, int *len)` — возвращает в строке `name` имя узла, на котором запущен вызвавший процесс. Возвращаемое имя должно идентифицировать конкретную часть аппаратного обеспечения, точный формат определяется реализацией. Это имя может совпадать или не совпадать с тем, которое может быть возвращено с помощью `gethostname`, `uname` или `sysinfo`. В переменной `len` возвращается количество символов в имени, не превышающее константы `MPI_MAX_PROCESSOR_NAME`.

Определение характеристик системного таймера.

```
#include <stdio.h>
#include "mpi.h"
#define NTIMES 100

int main(int argc, char **argv) {
    double time_start, time_finish, tick;
    int rank, i, len;
    char name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(name, &len);
    tick = MPI_Wtick();time_start = MPI_Wtime();
    for (i = 0; i<NTIMES; i++){
        time_finish = MPI_Wtime();
    }
   printf ("processor %s, prcs %d: tick= %lf, time=%lf\n",
name, rank, tick, (time_finish-time_start)/NTIMES);
   MPI_Finalize();
}
```

Передача сообщений.
Операции типа точка-точка.

В операциях типа точка-точка участвуют два процесса, один является отправителем сообщения, другой – получателем. Процесс-отправитель должен вызвать одну из процедур передачи данных и явно указать номер процесса-получателя в некотором коммуникаторе, а процесс-получатель должен вызвать одну из процедур приема с указанием того же коммуникатора. Он может не знать точный номер процесса-отправителя в данном коммуникаторе. Все процедуры делятся на два класса: процедуры с блокировкой и процедуры без блокировки (асинхронные). Процедуры обмена с блокировкой приостанавливают работу процесса до выполнения некоторого условия, а возврат из асинхронных процедур происходит немедленно после инициализации соответствующей коммуникационной операции.

Прием и передача сообщений с блокировкой.

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

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

Перечисление MPI_Datatype.

MPI_INT — int
MPI_SHORT — short
MPI_LONG — long
MPI_FLOAT — float
MPI_DOUBLE — double
MPI_CHAR — char
MPI_BYTE — 8 бит
MPI_PACKED — тип для упакованных данных.

Прием и передача сообщений с блокировкой.

Если при приеме сообщения пользователя не интересует заполнение структуры `status`, то вместо соответствующего аргумента можно указать предопределенную константу `MPI_STATUS_IGNORE`. Это также позволит сэкономить немного времени, требуемого на запись соответствующих полей. Вместо аргументов `source` и `msgtag` можно использовать константы:

`MPI_ANY_SOURCE` — признак того, что подходит сообщение от любого процесса;

`MPI_ANY_TAG` — признак того, что подходит сообщение с любым идентификатором.

Прием и передача сообщений с блокировкой. Пример для взаимодействия двух процессов.

```
#include "mpi.h"
#include <stdio.h>
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_FLOAT, 1, 5, MPI_COMM_WORLD);
      MPI_Recv(&a, 1, MPI_FLOAT, 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);
   }
   printf("process %d a = %f, b = %f\n", rank, a, b);
   MPI_Finalize();
}
```

Прием и передача сообщений с блокировкой. Обмен сообщениями четных и нечетных процессов.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
   int size, rank, a, b;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   a = rank;
   b = -1;
   if((rank%2) == 0){
      if(rank<size-1)
         MPI_Send(&a, 1, MPI_INT, rank+1, 5, MPI_COMM_WORLD);
   }
   else
      MPI_Recv(&b, 1, MPI_INT, rank-1, 5, MPI_COMM_WORLD,
&status);
   printf("process %d a = %d, b = %d\n", rank, a, b);
   MPI_Finalize();
}
```