

Лекция 3

MPI

Стандарт **MPI** (**MPI** –Message Passing Interface).

На кластере реализован **OpenMPI**. Это программная реализация стандарта интерфейса передачи сообщений (**MPI**) с открытым исходным кодом.

<http://www.open-mpi.org>

В вычислительных системах с **распределенной памятью** процессоры работают независимо друг от друга. Для организации параллельных вычислений в таких условиях необходимо иметь возможность *распределять* вычислительную нагрузку и *организовать* информационное взаимодействие (*передачу данных*) между процессорами (ядрами).

Решение этих задач и обеспечивает интерфейс передачи данных **MPI**. **MPI** работает и на системах с **общей памятью**!

В общем плане, для распределения вычислений между процессорами необходимо проанализировать алгоритм решения задачи, выделить информационно **независимые** фрагменты вычислений, провести их программную реализацию и затем разместить полученные части программы на разных процессорах (ядрах).

MPI

В рамках MPI (как и в случае OpenMP) принят более простой подход – *разрабатывается одна программа и эта единственная программа запускается одновременно на выполнение на всех имеющихся процессорах (ядрах)*! При этом для того, чтобы избежать идентичности вычислений на разных процессорах, можно,

- во-первых, подставлять **разные данные** для программы на **разных процессорах**,
- во-вторых, в MPI (как и в OpenMP) имеются средства для идентификации процесса, и тем самым, предоставляется возможность организовать различия в вычислениях в разных процессах (процессорах).

Подобный способ организации параллельных вычислений получил наименование модели **SPMD**, а в случае MPI эту модель часто называют: "одна программа множество процессов" (*single program multiple processes* - **SPMP**).

MPI

Для организации информационного взаимодействия между процессорами в самом минимальном варианте достаточно операции приема и передачи данных (при этом, конечно, должна существовать техническая возможность коммуникации между процессорами – *каналы* или *линии связи*) В MPI существует целое множество операций передачи данных. Они реализуют практически все возможные коммуникационные операции. Именно это является наиболее сильной стороной MPI.

Итак, во-первых, MPI - это **стандарт**, которому должны удовлетворять средства организации передачи сообщений.

Во-вторых, MPI – это **программные средства**, которые обеспечивают возможность передачи сообщений и при этом соответствуют всем требованиям стандарта MPI. Так, по стандарту, эти программные средства должны быть **организованы в виде библиотек** программных модулей (*библиотеки MPI*) и должны быть доступны для наиболее широко используемых алгоритмических языков C и Fortran.

МРІ. Понятие параллельной программы

Под **параллельной программой** в рамках МРІ понимается множество одновременно выполняемых **процессов**.

Процессы могут выполняться на **разных процессорах**, но на одном процессоре могут располагаться и несколько процессов (в этом случае их исполнение осуществляется в режиме деления времени).

В предельном случае для выполнения параллельной программы может использоваться **один процессор** — как правило, такой способ применяется для начальной проверки правильности параллельной программы.

Каждый процесс параллельной программы порождается на основе копии одного и того же программного кода (*модель SPM*). Данный программный код, представленный в виде исполняемой программы, должен быть доступен в момент запуска параллельной программы на всех используемых процессорах.

MPI. Понятие параллельной программы

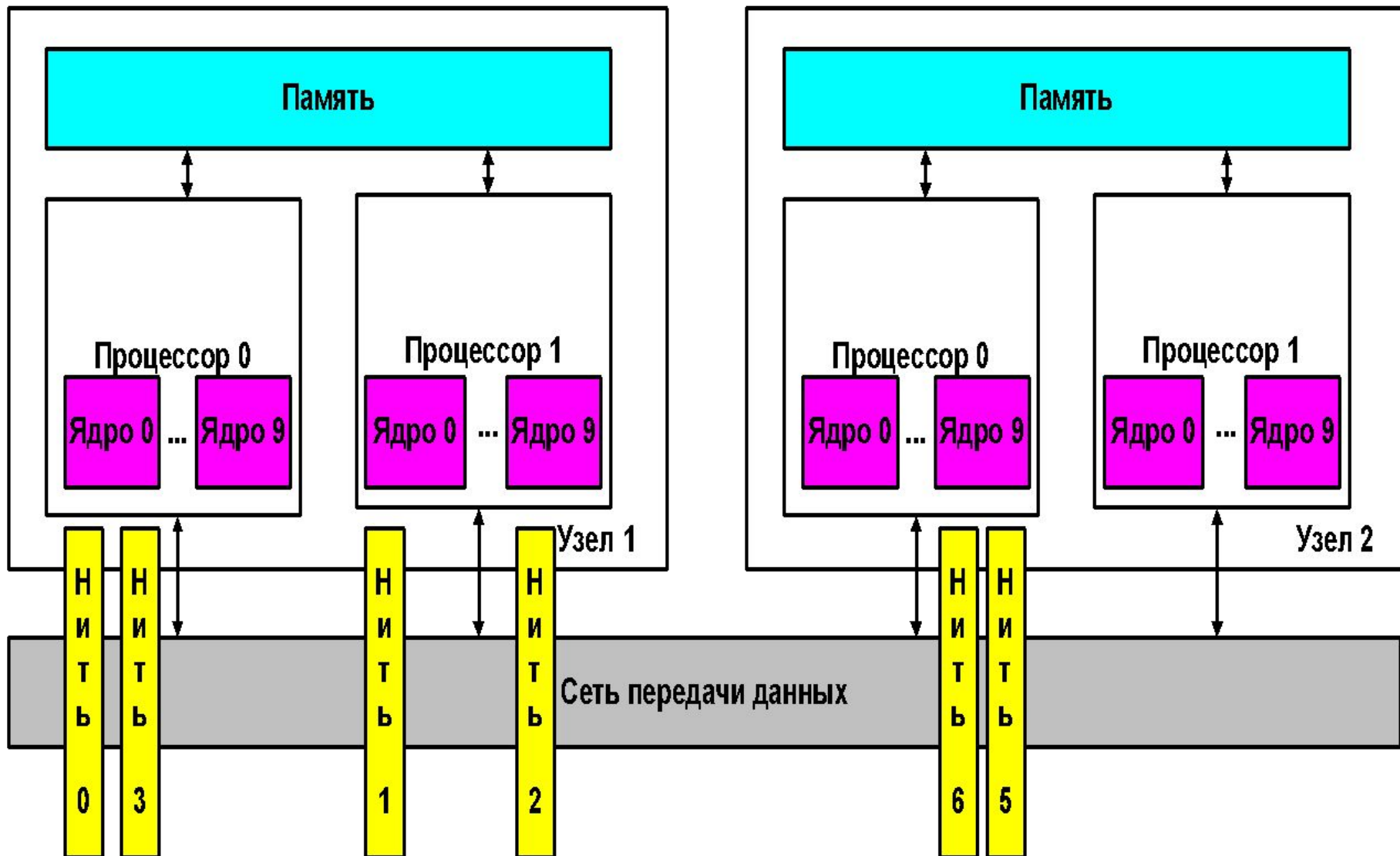
Исходный программный код для исполняемой программы разрабатывается на алгоритмических языках С или Fortran с использованием той или иной реализации библиотеки MPI.

Количество процессов и число используемых процессоров **определяется в момент запуска** параллельной программы средствами среды исполнения MPI-программ и в ходе вычислений **меняться не может**. В этом одно из главных отличий от **OpenMP**, в которой реализована «пульсирующая» модель типа разветвление – слияние.

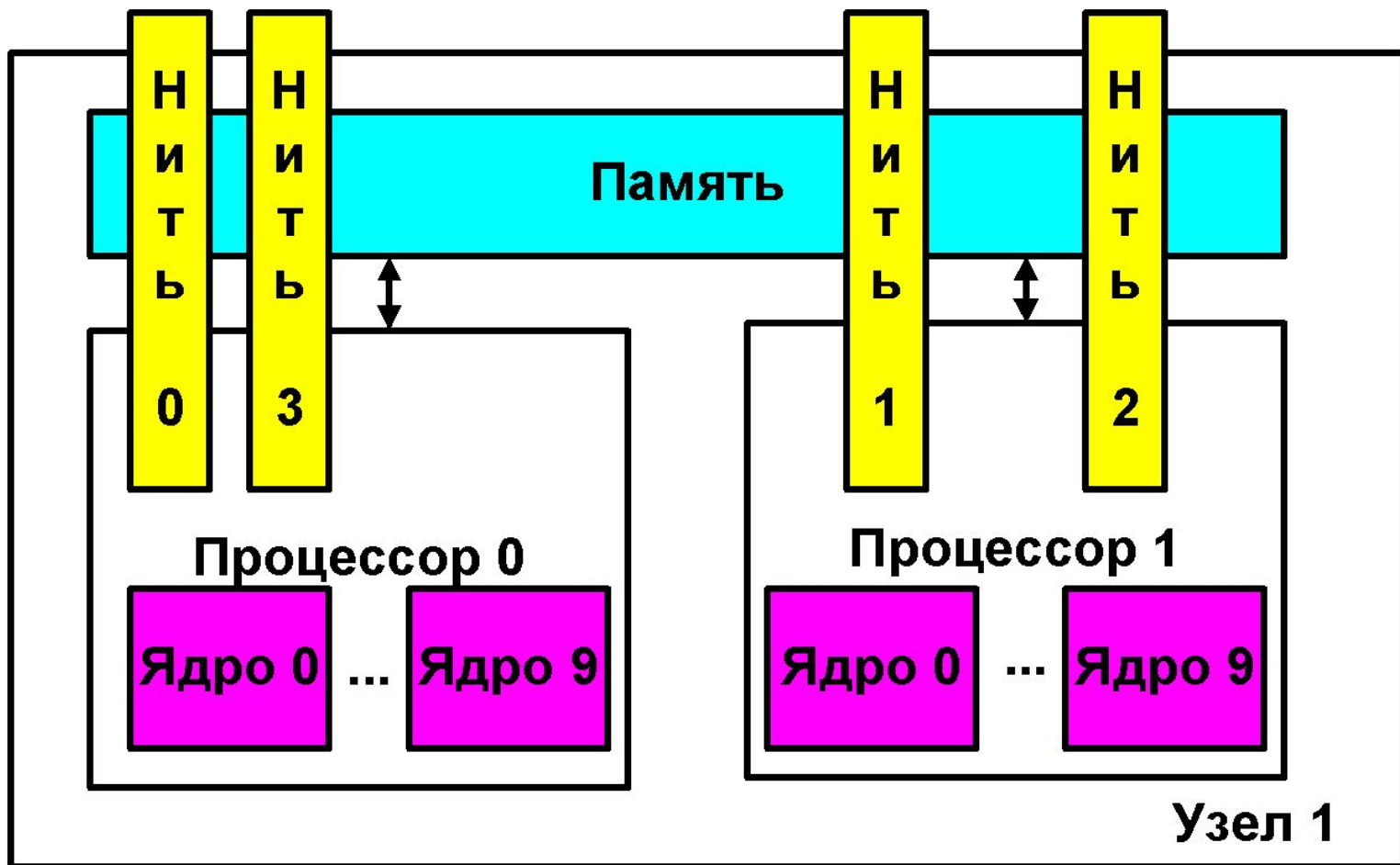
Все процессы программы последовательно перенумерованы от 0 до $p-1$, где p есть общее количество процессов.

Номер процесса именуется **рангом (*rank*)** процесса.

МРІ. Понятие параллельной программы



OpenMP. Параллельная программа



МРІ. Операции передачи данных

Основу МРІ составляют операции передачи сообщений. Среди предусмотренных в составе МРІ функций различаются **парные** (*point-to-point*) операции между двумя процессами и **коллективные** (*collective*) коммуникационные действия для одновременного взаимодействия нескольких процессов.

Для выполнения парных операций могут использоваться разные *режимы передачи*, среди которых синхронный, блокирующий и др.

Стандарт МРІ предусматривает необходимость реализации большинства основных коллективных операций передачи данных.

МРІ. Понятие коммунікаторов

Процессы параллельной программы объединяются в *группы*. Под *коммунікатором* в МРІ понимается специально создаваемый служебный объект, объединяющий в своем составе группу процессов и ряд дополнительных параметров (*контекст*), используемых при выполнении операций передачи данных.

Как правило, парные операции передачи данных выполняются для процессов, принадлежащих **одному и тому же коммунікатору**. Коллективные операции применяются одновременно для всех процессов коммунікатора.

В ходе вычислений могут создаваться новые и удаляться существующие группы процессов и коммунікаторы. Один и тот же процесс может принадлежать разным группам и коммунікаторам. Все имеющиеся в параллельной программе процессы входят в состав создаваемого по умолчанию коммунікатора с идентификатором **MPI_COMM_WORLD**.

При необходимости передачи данных между процессами из разных групп необходимо создавать глобальный коммунікатор (*intercommunicator*).

MPI. Типы данных

При выполнении операций передачи сообщений для указания 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 алгоритмических языках C и Fortran.

Кроме того, в MPI имеются возможности для создания новых *производных типов* данных для более точного и краткого описания содержимого пересылаемых сообщений.

MPI. Инициализация и завершение MPI программ

Первой вызываемой функцией MPI должна быть функция:

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

для инициализации среды выполнения MPI-программы.

Параметрами функции являются количество аргументов в командной строке и текст самой командной строки, передаваемые в основную программу **main**.

Последней вызываемой функцией MPI обязательно должна являться функция:

int MPI_Finalize (void);

MPI. 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 ( &argc, &argv );
<программный код с использованием MPI функций>
MPI_Finalize();
<программный код без использования MPI функций>
return 0; }
```

1. Файл **mpi.h** содержит определения именованных констант, прототипов функций и типов данных библиотеки MPI,
2. Функции **MPI_Init** и **MPI_Finalize** являются обязательными и должны быть выполнены (**и только один раз!**) каждым процессом параллельной программы,

MPI. 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 *номера (ранга) процесса* используется функция:

int MPI_Comm_rank (MPI_Comm comm, int *rank).

Как правило, вызов функций **MPI_Comm_size** и **MPI_Comm_rank** выполняется сразу после **MPI_Init**.

Число процессов (**size**) задается **средой выполнения MPI при запуске программы** и не может быть изменено при выполнении программы! (В отличие от OpenMP).

MPI_Comm определяет специальный тип объекта-коммуникатора MPI, **comm** — имя коммуникатора, в который группируются все процессы программы.

MPI. Определение количества и ранга процессов

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

1. Коммуникатор **MPI_COMM_WORLD** создается по умолчанию и представляет все процессы выполняемой параллельной программы,
2. Ранг, получаемый при помощи функции **MPI_Comm_rank**, является рангом процесса, выполнившего вызов этой функции, т.е. переменная **rank** будет принимать различные значения в разных процессах.

МРІ. 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,
- **MPI_Datatype** – тип данных MPI

MPI. Базовые типы данных

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

MPI. Передача сообщений

1. Отправляемое сообщение определяется через указание блока памяти (буфера), в котором это сообщение располагается. Используемая для указания буфера тройка (**buf, count, type**) входит в состав параметров практически всех функций передачи данных.
2. Процессы, между которыми выполняется передача данных, в обязательном порядке должны принадлежать коммуникатору, указываемому в функции **MPI_Send**,
3. Параметр **tag** используется только при необходимости различения передаваемых сообщений, в противном случае в качестве значения параметра может быть использовано произвольное целое неотрицательное число, лежащее в диапазоне от 0 до 32767.

MPI. Передача сообщений

Сразу же после завершения функции **MPI_Send** процесс-отправитель может начать повторно использовать буфер памяти, в котором располагалось отправляемое сообщение.

Вместе с этим, следует понимать, что в момент завершения функции **MPI_Send** состояние самого пересылаемого сообщения может быть совершенно различным - сообщение может располагаться в процессе-отправителе, может находиться в процессе передачи, может храниться в процессе-получателе или же может быть принято процессом-получателем при помощи функции **MPI_Recv**.

Тем самым, завершение функции **MPI_Send** означает лишь, что операция передачи начала выполняться и пересылка сообщения будет рано или поздно будет выполнена.

MPI. 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, назначение

каждого отдельного параметра соответствует описанию в **MPI_Send**,

- **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.

MPI. Прием сообщений

1. 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.
2. При необходимости приема сообщения от любого процесса-отправителя для параметра **source** может быть указано значение **MPI_ANY_SOURCE**.
3. При необходимости приема сообщения с любым тегом для параметра **tag** может быть указано значение **MPI_ANY_TAG**.
4. **MPI_Status** – специальная структура MPI, которая содержит три поля:
 - **status.MPI_SOURCE** – ранг процесса-отправителя принятого сообщения,
 - **status.MPI_TAG** - тег принятого сообщения.
 - **status.MPI_Error** - код ошибки.

MPI. Прием сообщений

Вызов функции **MPI_Recv** не должен согласовываться со временем вызова соответствующей функции передачи сообщения **MPI_Send** — прием сообщения может быть инициирован до момента, в момент или после момента начала отправки сообщения.

По завершении функции **MPI_Recv** в заданном буфере памяти будет располагаться принятое сообщение. Принципиальный момент здесь состоит в том, что 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**.

Рассмотренный выше набор из 6 функций MPI оказывается достаточным для разработки параллельных программ!

MPI_Init, MPI_Finalize, MPI_Comm_size, MPI_Comm_rank, MPI_Send, MPI_Recv.

MPI. Определение времени

Получение времени **выполнения** текущего момента выполнения программы обеспечивается при помощи функции:

double MPI_Wtime(void),

результат вызова которой есть количество секунд, прошедшее от некоторого определенного момента времени в прошлом. Этот момент времени в прошлом, от которого происходит отсчет секунд, может зависеть от среды реализации библиотеки MPI и, тем самым, для ухода от такой зависимости функцию **MPI_Wtime** следует использовать только для определения длительности выполнения тех или иных фрагментов кода параллельных программ. Возможная схема применения функции **MPI_Wtime** может состоять в следующем:

double t1, t2, dt;

t1 = MPI_Wtime();

...

t2 = MPI_Wtime();

dt = t2 - t1;

Точность измерения времени также может зависеть от среды выполнения параллельной программы. Для определения текущего значения точности может быть использована функция:

double MPI_Wtick(void),

позволяющая определить время в секундах между двумя последовательными показателями времени аппаратного таймера используемой компьютерной системы.

MPI. Коллективные операции пересылки

Для выполнения **коммуникационных** *коллективных операций*, в которых принимают участие все процессы коммуникатора, в MPI предусмотрен специальный набор функций.

MPI. Передача данных от одного процесса всем процессам программы

int MPI_Bcast (void *buf, int count, MPI_Datatype type, int root, MPI_Comm comm),

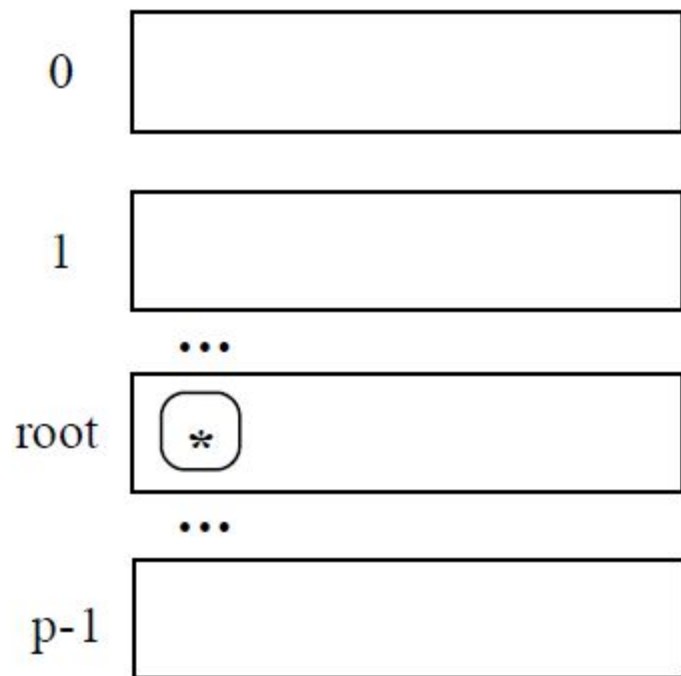
где

- **buf, count, type** – буфер памяти с отправляемым сообщением (для процесса с рангом **root**), и для приема сообщений для всех остальных процессов,
- **root** - ранг процесса, выполняющего рассылку данных,
- **comm** - коммуникатор, в рамках которого выполняется передача данных.

Функция **MPI_Bcast** осуществляет рассылку данных из буфера **buf**, содержащего **count** элементов типа **type** с процесса, имеющего номер **root**, всем процессам, входящим в коммуникатор **comm**.

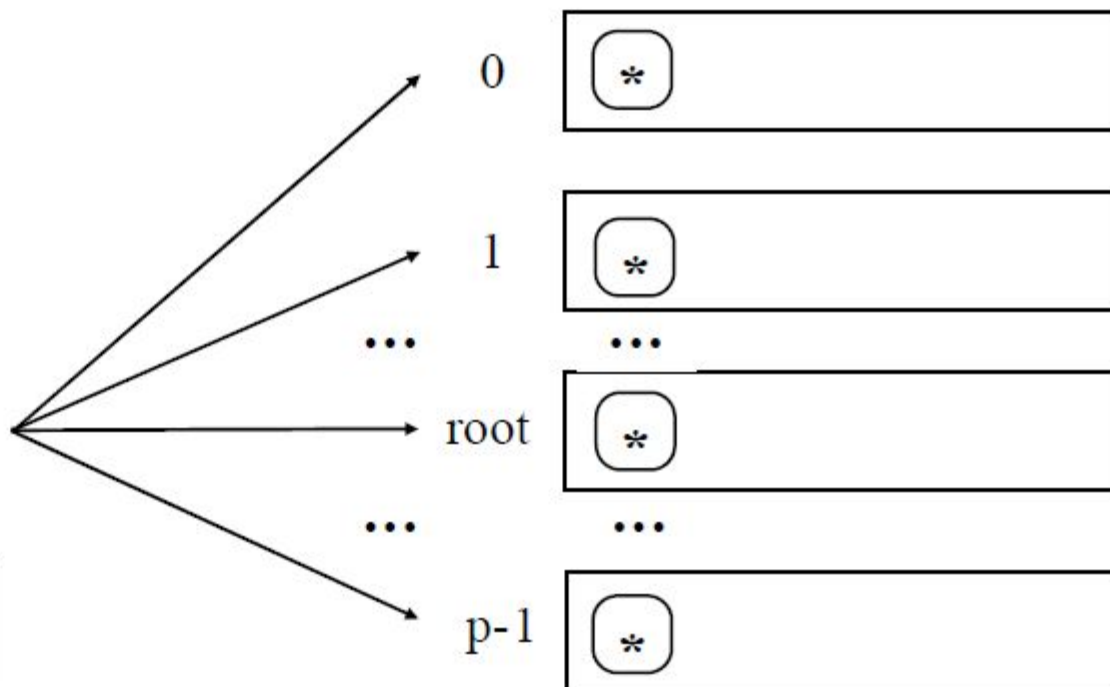
МРІ. Передача данных от одного процесса всем процессам программы

процессы



а) до начала операции

процессы



б) после завершения операции

MPI. Передача данных от одного процесса всем процессам программы

Функция **MPI_Bcast** определяет коллективную операцию и, тем самым, при выполнении необходимых рассылок данных вызов функции **MPI_Bcast** должен быть осуществлен всеми процессами указываемого коммутатора.

Указываемый в функции **MPI_Bcast** буфер памяти имеет различное назначение в разных процессах. Для процесса с рангом **root**, с которого осуществляется рассылка данных, в этом буфере должно находиться рассылаемое сообщение. Для всех остальных процессов указываемый буфер предназначен для приема передаваемых данных.

MPI. Передача данных от всех процессов одному процессу. Операции редукции

В этой операции над собираемыми значениями осуществляется та или иная обработка данных (именуется *операцией редукции данных*). Для наилучшего выполнения действий, связанных с редукцией данных, в MPI предусмотрена функция:

```
int MPI_Reduce (void *sendbuf, void *recvbuf, int count,  
MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm),
```

где

- **sendbuf** - буфер памяти с отправляемым сообщением,
- **recvbuf** – буфер памяти для результирующего сообщения (только для процесса с рангом **root**),
- **count** - количество элементов в сообщениях,
- **type** – тип элементов сообщений,
- **op** - операция, которая должна быть выполнена над данными,
- **root** - ранг процесса, на котором должен быть получен результат,
- **comm** - коммуникатор, в рамках которого выполняется операция.

МРІ. Передача данных от всех процессов одному процессу. Операции редукции

Операция	Описание
MPI_MAX	Определение максимального значения
MPI_MIN	Определение минимального значения
MPI_SUM	Определение суммы значений
MPI_PROD	Определение произведения значений
MPI_BAND	Выполнение логической операции "И" над значениями сообщений
MPI_BAND	Выполнение битовой операции "И" над значениями сообщений
MPI_LOR	Выполнение логической операции "ИЛИ" над значениями сообщений
MPI_BOR	Выполнение битовой операции "ИЛИ" над значениями сообщений
MPI_LXOR	Выполнение логической операции исключающего "ИЛИ" над значениями сообщений
MPI_BXOR	Выполнение битовой операции исключающего "ИЛИ" над значениями сообщений
MPI_MAXLOC	Определение максимальных значений и их индексов
MPI_MINLOC	Определение минимальных значений и их индексов

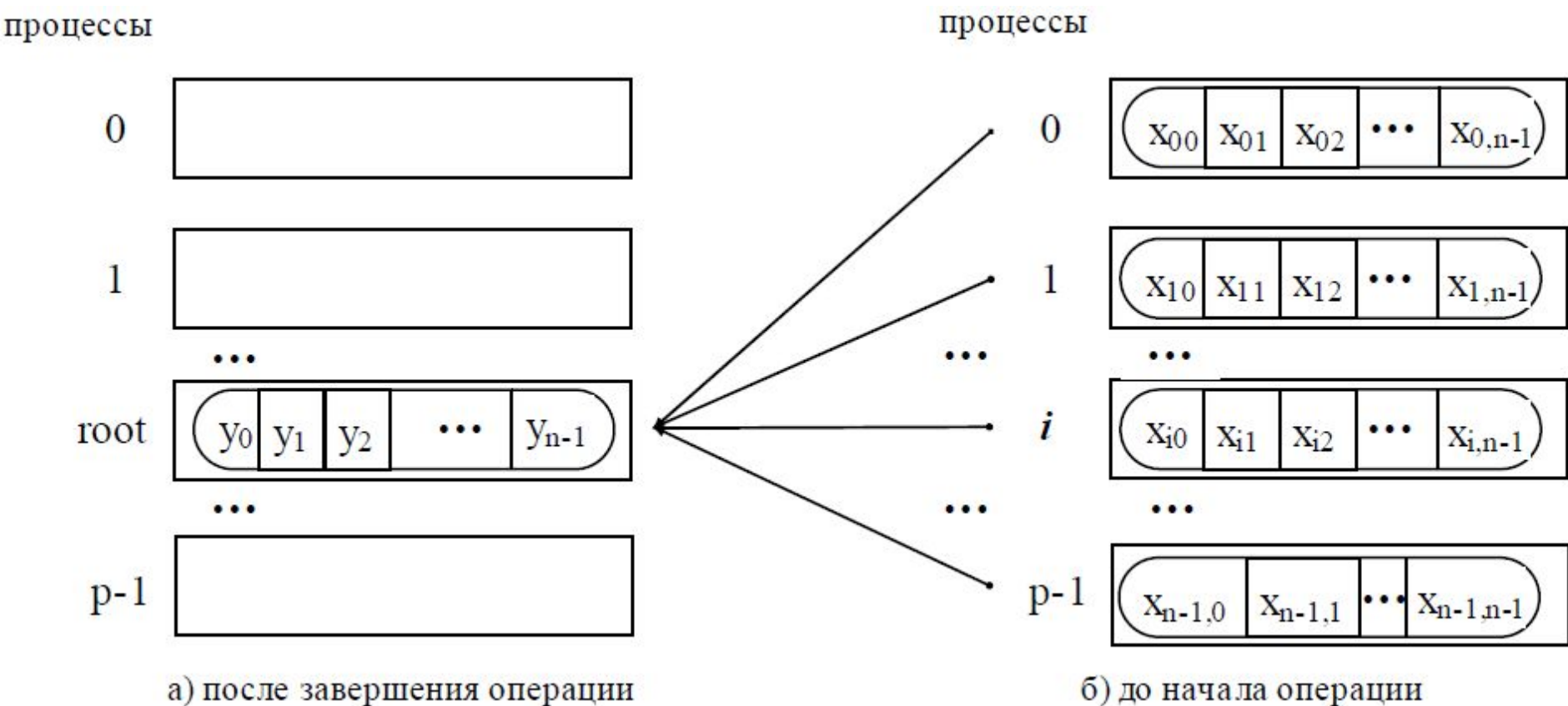
MPI. Передача данных от всех процессов одному процессу. Операции редукции

Элементы получаемого сообщения на процессе **root** представляют собой результаты обработки соответствующих элементов передаваемых процессами сообщений:

$$y_j = \bigotimes_{i=0}^{n-1} x_{ij}, \quad 0 \leq j < n,$$

где $\otimes$ есть операция, задаваемая при вызове функции **MPI_Reduce**.

МРІ. Передача данных от всех процессов одному процессу. Операции редукции



$$y_j = \bigotimes_{i=0}^{n-1} x_{ij}, 0 \leq j < n,$$

MPI. Передача данных от всех процессов одному процессу. Операции редукции

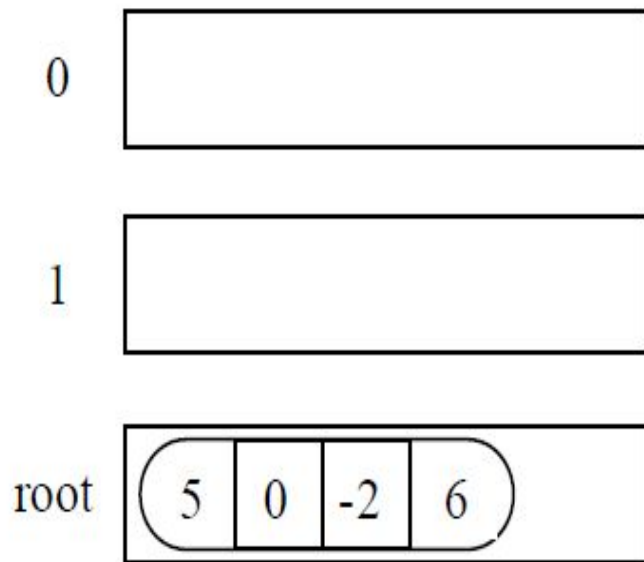
Функция **MPI_Reduce** определяет коллективную операцию и, тем самым, вызов функции должен быть выполнен всеми процессами указываемого коммуникатора, все вызовы функции должны содержать одинаковые значения параметров **count**, **type**, **op**, **root**, **comm**.

Передача сообщений должна быть выполнена всеми процессами, результат операции будет получен только процессом с рангом **root**.

Выполнение операции редукции осуществляется над отдельными элементами передаваемых сообщений. Так, например, если сообщения содержат по два элемента данных и выполняется операция суммирования **MPI_SUM**, то результат также будет состоять из двух значений, первое из которых будет содержать сумму первых элементов всех отправленных сообщений, а второе значение будет равно сумме вторых элементов сообщений соответственно.

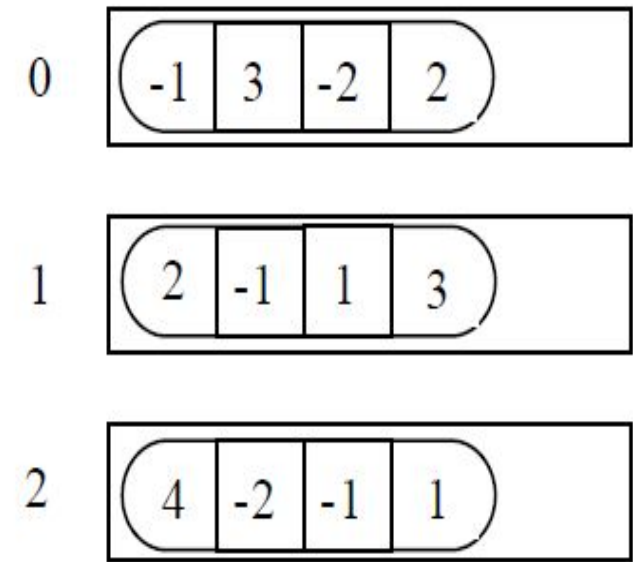
МРІ. Передача данных от всех процессов одному процессу. Операции редукции

процессы



а) после завершения операции

процессы



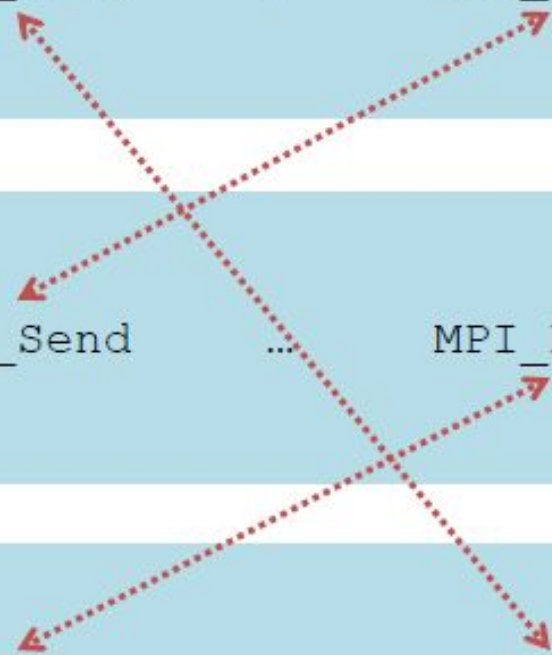
б) до начала операции

MPI. Понятие параллельной программы

MPI_Init MPI_Send ... MPI_Recv MPI_Finalize

MPI_Init MPI_Send ... MPI_Recv MPI_Finalize

MPI_Init MPI_Send ... MPI_Recv MPI_Finalize



МРІ. Что осталось «за кадром»?

1. Модификации обмена сообщениями точка-точка.
2. Модификации коллективных обменов сообщениями.
3. Работа с коммутаторами, с данными
4. Виртуальные топологии.

МРІ. Компіляція і запуск

Компіляція для мови Си:

mpicc <ім'я вихідного файлу.c> -o <ім'я виконуваного файлу>

Запуск без системи SLURM на одному вузлі:

mpirun -n <кількість процесів> <ім'я виконуваного файлу>

Запуск в системі SLURM на одному вузлі:

salloc -N1 -n<кількість процесів> mpirun <ім'я виконуваного файлу>

параметр -N задає кількість вузлів.

Запуск в системі SLURM на двох вузлах:

salloc -N2 -n<кількість процесів> mpirun -mca oob_tcp_if_include bond0 <ім'я виконуваного файлу>

Рядок **-mca oob_tcp_if_include bond0** «підключає» інтерфейс між вузлами.

MPI. Примеры программ

Программа 1. Определение версии MPI

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
    int version, sub_version;

    MPI_Init(&argc, &argv);

    MPI_Get_version(&version, &sub_version);

    printf("MPI Version %d.%d\n", version, sub_version);

    MPI_Finalize();
    return 0;
}
```

MPI. Примеры программ

Программа 2. Определение числа процессов и номера процесса

//Число процессов задается параметрами среды выполнения

MPI при запуске

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

```
#include "mpi.h"
```

```
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);
```

```
    printf ("process %d, size %d\n", rank, size);
```

```
    MPI_Finalize();
```

```
    return (0);
```

```
}
```

MPI. Примеры программ

Программа 3. Определение имени узла (узлов)

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank, size, len;
    char name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Get_processor_name (name, &len); /*Процедура возвращает в
строке name имя узла, на котором запущен вызвавший процесс. В
переменной len возвращается количество символов в имени, не
превышающее
значения константы MPI_MAX_PROCESSOR_NAME*/
    printf ("Hello. I am %d of %d on %s\n", rank, size, name);
    MPI_Finalize();
    return 0; }
```

MPI. Примеры программ

Программа 4. Системный таймер. Замер времени. Точность таймера

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
    double start_time, end_time, tick;
    MPI_Init(&argc, &argv);
    start_time = MPI_Wtime();
    sleep (1);
    end_time = MPI_Wtime();
    tick = MPI_Wtick();
    printf ("Time to measure %E\n", end_time-start_time);
    printf ("Accuracy of timer %E\n", tick);
    MPI_Finalize();
    return 0;
}
```

MPI. Примеры программ

Программа 5. Обмен сообщениями двух процессов

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.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);
```

MPI. Примеры программ

Программа 5. Продолжение

```
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();
return (0);
}
```

MPI. Примеры программ

Программа 6. Обмен сообщениями четных и нечетных процессов

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
    int size, rank, b, c, d;
    MPI_Status status;
    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
```

MPI. Примеры программ

Программа 6. Продолжение

```
b = -1;  c = 0;  d = 1;
    if((rank%2) == 0){
        if(rank<size-1) {
MPI_Send(&b, 1, MPI_INT, rank+1, 5, MPI_COMM_WORLD);
MPI_Recv(&c, 1, MPI_INT, rank+1, 5, MPI_COMM_WORLD,
&status);}
        else c=1;  }
    else    {
MPI_Recv(&c, 1, MPI_INT, rank-1, 5, MPI_COMM_WORLD,
&status);
MPI_Send(&d, 1, MPI_INT, rank-1, 5, MPI_COMM_WORLD);}
    printf ("process %d, b = %d, c = %d, d = %d\n  ", rank, b, c, d);
    MPI_Finalize();
return (0);
}
```

MPI. Примеры программ

Программа 7.

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{int i;
 MPI_Init(&argc, &argv);
 if (i == MPI_SUCCESS)
 printf("Successful initialization with code %d\n", i);
 else
 printf("Initialization failed with error code %d\n", i);
 MPI_Finalize();
 if (i == MPI_SUCCESS)
 printf("Successful MPI_Finalize() with code %d\n", i);
 else
 printf("MPI_Finalize() failed with error code %d\n", i);
 return (0);
}
```

MPI. Примеры программ

Программа 8. Моделирование тупика

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
    int rank, b, c;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

MPI. Примеры программ

Программа 8. Продолжение

```
b = -10;  c = 10;
    if(rank == 0){

        MPI_Send(&b, 1, MPI_INT, 1, 5, MPI_COMM_WORLD);
        MPI_Recv(&c, 1, MPI_INT, 1, 5, MPI_COMM_WORLD,
&status); }
    if(rank == 1)
    {
        MPI_Recv(&c, 1, MPI_INT, 0, 5, MPI_COMM_WORLD,
&status);
        MPI_Send(&b, 1, MPI_INT, 0, 5, MPI_COMM_WORLD);
    }
    printf("process %d, b = %d, c = %d\n ", rank, b, c);
    MPI_Finalize();
return (0);
}
```

Программа 9. Программа с приветом

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[]){
int size, rank, RecvRank;
MPI_Status Status;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if ( rank == 0 ){
// Действия, выполняемые только процессом с рангом 0
printf ("\n Hello from process %3d", rank);
for ( int i=1; i<size; i++ ) {
MPI_Recv(&RecvRank, 1, MPI_INT, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
printf("\n Hello from process %3d", RecvRank); }}
else // Сообщение, отправляемое всеми процессами,
// кроме процесса с рангом 0
MPI_Send(&rank,1,MPI_INT,0,0,
MPI_COMM_WORLD); MPI_Finalize();
return 0; }
```

Каждый процесс определяет свой ранг, после чего действия в программе разделяются. 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. Порядок приема сообщений заранее не определен и зависит от условий выполнения параллельной программы (более того, этот порядок может изменяться от запуска к запуску).

Возможный вариант результатов печати процесса 0 может состоять в следующем (для параллельной программы из четырех процессов):

Hello from process 0

Hello from process 2

Hello from process 1

Hello from process 3

Такой "плавающий" вид получаемых результатов существенным образом усложняет разработку, тестирование и отладку параллельных программ, т.к. в этом случае исчезает один из основных принципов программирования – повторяемость выполняемых вычислительных экспериментов.

Как правило, если это не приводит к потере эффективности, желательно обеспечивать однозначность расчетов и при использовании параллельных вычислений. Так, для рассматриваемого простого примера можно восстановить постоянство получаемых результатов при помощи задания ранга процесса-отправителя в операции приема сообщения:

`MPI_Recv(&RecvRank, 1, MPI_INT, i, MPI_ANY_TAG, MPI_COMM_WORLD, &Status).`

Указание ранга процесса-отправителя регламентирует порядок приема сообщений, и, как результат, строки печати будут появляться строго в порядке возрастания рангов процессов (повторим, что такая регламентация в отдельных ситуациях может приводить к замедлению выполняемых параллельных вычислений).

MPI

Для разделения фрагментов кода между процессами обычно используется подход, примененный в только что рассмотренной программе - при помощи функции **MPI_Comm_rank** определяется ранг процесса, а затем в соответствии с рангом выделяются необходимые для процесса участки программного кода.

Наличие в одной и той же программе фрагментов кода разных процессов также значительно усложняет понимание и, в целом, разработку MPI-программы. Как результат, можно рекомендовать при увеличении объема разрабатываемых программ выносить программный код разных процессов в отдельные программные модули (функции). Общая схема MPI программы в этом случае будет иметь вид:

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
if ( rank == 0 ) DoProcess0();  
else if ( rank == 1 ) DoProcess1();  
else if ( rank == 2 ) DoProcess2();
```

Во многих случаях, как и в рассмотренном примере, выполняемые действия являются отличающимися только для процесса с рангом 0. В этом случае общая схема MPI программы принимает более простой вид:

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
if ( rank == 0 ) DoManagerProcess();  
else DoWorkerProcesses();
```

MPI и OpenMP

Плюсы MPI

- Переносимость для систем с общей и распределенной памятью
- Масштабируемость при увеличении узлов

Минусы MPI

- Сложность разработки и отладки
- Высокая латентность
- Явные коммуникации
- Сложность балансировки загрузки

Плюсы OpenMP

- Простая реализация параллелизма
- Низкая латентность
- Неявные коммуникации
- Динамическая балансировка загрузки

Минусы OpenMP

- Переносимость только для систем с общей памятью
- Масштабируемость в пределах одного узла
- Отсутствие возможности задать порядок нитей

MPI + OpenMP

Архитектура MPI+OpenMP: плюсы

1. Удобное применение для кластеров с SMP-узлами:
 - MPI — между узлами: избегаем накладных расходов на MPI-коммуникации внутри узла;
 - OpenMP — внутри узла: получаем передачу сообщений большего размера за меньшее время и динамическую балансировку загрузки.
2. Потенциальная возможность получить большее ускорение, чем "чистый" MPI или "чистый" OpenMP.

Архитектура MPI+OpenMP: минусы

1. Меньшая масштабируемость OpenMP.
2. Возможность тупиков в MPI.

MPI + OpenMP

Компиляция MPI+OpenMP-программ

mpicc <имя исходного файла.c> -fopenmp -o <имя исполняемого файла>