Технология программирования MPI (4)

Антонов Александр Сергеевич,
к.ф.-м.н., вед.н.с. лаборатории Параллельных информационных технологий НИВЦ МГУ

Летняя суперкомпьютерная академия
Москва, 2015

MPI

Пересылка разнотипных данных

MPI

Сообщение – массив однотипных данных, расположенных в последовательных ячейках памяти.

Для пересылки разнотипных данных можно использовать:

- Производные типы данных
- Упаковку данных

MPI

Производные типы данных создаются во время выполнения программы с помощью подпрограмм-конструкторов.

Создание типа:

- Конструирование типа
- Регистрация типа

MPI

Производный тип данных характеризуется последовательностью базовых типов и набором значений смещения относительно начала буфера обмена.

Смещения могут быть как положительными, так и отрицательными, не требуется их упорядоченность.

MPI

```
int MPI_Type_contiguous(int
count, MPI_Datatype type,
MPI_Datatype *newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** последовательно расположенных элементов базового типа данных **type**.

```
MPI_Type_contiguous(5, MPI_INT, &newtype);
```

MPI

```
int MPI_Type_vector(int count,
int blocklen, int stride,
MPI_Datatype type, MPI_Datatype
*newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklen** элементов базового типа данных **type**. Следующий блок начинается через **stride** элементов после начала предыдущего.

MPI

```
count=2;
blocklen=3;
stride=5;
MPI_Type_vector(count, blocklen, stride,
MPI_DOUBLE, &newtype);
```

Создание нового типа данных (тип элемента, количество элементов от начала буфера):

{(**MPI_DOUBLE**, 0), (**MPI_DOUBLE**, 1), (**MPI_DOUBLE**, 2),
(**MPI_DOUBLE**, 5), (**MPI_DOUBLE**, 6), (**MPI_DOUBLE**, 7)}

MPI

```
int MPI_Type_create_hvector(int
count, int blocklen, MPI_Aint
stride, MPI_Datatype type,
MPI_Datatype *newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklen** элементов базового типа данных **type**. Следующий блок начинается через **stride** байт после начала предыдущего.

MPI

```
int
MPI_Type_create_indexed_block(in
t count, int blocklen, int
displs[], MPI_Datatype type,
MPI_Datatype *newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklen** элементов базового типа данных **type**. Смещения блоков с начала буфера посылки в количестве элементов базового типа данных **type** задаются в массиве **displs**.

MPI

```
int MPI_Type_indexed(int count,
int *blocklens, int *displs,
MPI_Datatype type, MPI_Datatype
*newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklens[i]** элементов базового типа данных. **i**-ый блок начинается через **displs[i]** элементов с начала буфера.

MPI

```
for(i=0; i<n; i++){
   blocklens[i]=n-i;
   displs[i]=(n+1)*i;
}
MPI_Type_indexed(n, blocklens, displs,
MPI_DOUBLE, &newtype)
```

Создание нового типа данных для описания верхнетреугольной матрицы.

MPI

```
int MPI_Type_create_hindexed(int
count, int *blocklens, MPI_Aint
*displs, MPI_Datatype type,
MPI_Datatype *newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklens[i]** элементов базового типа данных. **i**-ый блок начинается через **displs[i]** байт с начала буфера.

MPI

```
int MPI_Type_create_struct(int count, int *blocklens, MPI_Aint *displs, MPI_Datatype *types, MPI_Datatype *newtype)
```

Создание структурного типа данных из **`count`** блоков по **`blocklens[i]`** элементов типа **`types[i]`**. **`i`**-ый блок начинается через **`displs[i]`** байт с начала буфера.

MPI

```
blocklens[0]=3;
blocklens[1]=2;
types[0]=MPI_DOUBLE;
types[1]=MPI_CHAR;
displs[0]=0;
displs[1]=24;
MPI_Type_create_struct(2, blocklens, displs,
types, &newtype);
```

Создание нового типа данных (тип элемента, количество байт от начала буфера):

{(**MPI_DOUBLE**, 0), (**MPI_DOUBLE**, 8), (**MPI_DOUBLE**, 16),
(**MPI_CHAR**, 24), (**MPI_CHAR**, 25)}

MPI

```
int MPI_Type_create_subarray(int ndims, int sizes[], int subsizes[], int starts[], int order, MPI_Datatype type, MPI_Datatype *newtype)
```

newtype задаёт **ndims**–мерный подмассив исходного **ndims**–мерного массива. **sizes** задает размеры по каждому измерению исходного массива, **subsizes** – размеры по каждому измерению выделяемого подмассива.

MPI

`starts` задаёт стартовые координаты каждого измерения выделяемого подмассива в исходном массиве. Все массивы индексируются с 0. Задаваемые значения не должны выводить подмассив за пределы исходного массива ни по одному из измерений. **`order`** задаёт порядок хранения элементов многомерного массива: **`MPI_ORDER_C`** (по строкам), **`MPI_ORDER_FORTRAN`** (по столбцам). **`type`** задаёт тип элементов массива.

MPI

```
double subarray[100][25];
MPI_Datatype newtype;
int sizes[2], subsizes[2], starts[2];
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
sizes[0] = 100;
sizes[1] = 100;
subsizes[0] = 100;
subsizes[1] = 25;
starts[0] = 0;
starts[1] = rank*subsizes[1];
MPI_Type_create_subarray(2, sizes, subsizes,
starts, MPI_ORDER_C, MPI_DOUBLE, &newtype);
```

MPI

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

Регистрация созданного производного типа данных **`datatype`**. После регистрации этот тип данных можно использовать в операциях обмена.

MPI

```
int MPI_Type_free(MPI_Datatype
*datatype)
```

Аннулирование производного типа данных **`datatype`**. **`datatype`** устанавливается в значение **`MPI_DATATYPE_NULL`**. Производные от **`datatype`** типы данных остаются. Предопределённые типы данных не могут быть аннулированы.

MPI

```
int MPI_Get_address(void
*location, MPI_Aint *address)
```

Определение абсолютного байт-адреса **`address`** размещения массива **`location`** в оперативной памяти компьютера. Адрес отсчитывается от некоторого базового адреса, значение которого содержится в системной константе **`MPI_BOTTOM`**.

MPI

```
blocklens[0] = 1;
blocklens[1] = 1;
types[0] = MPI_DOUBLE;
types[1] = MPI_CHAR;
MPI_Get_address(dat1, &displs[0]);
MPI_Get_address(dat2, &displs[1]);
MPI_Type_create_struct(2, blocklens, displs,
types, &newtype);
MPI_Type_commit(&newtype);
MPI_Send(MPI_BOTTOM, 1, newtype, dest, tag,
MPI_COMM_WORLD);
```

MPI

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

Определение размера типа **`datatype`** в байтах (объёма памяти, занимаемого одним элементом данного типа).

MPI

```
int
MPI_Type_get_extent(MPI_Datatype
datatype, MPI_Aint *lb, MPI_Aint
*extent)
```

Для элемента типа данных **datatype** определяет смещение от начала буфера данных нижней границы **lb** и диапазон **extent** (разницу между верхней и нижней границами) в байтах.

MPI

```
int MPI_Pack(void *inbuf, int
incount, MPI_Datatype datatype,
void *outbuf, int outsize, int
*position, MPI_Comm comm)
```

Упаковка **incount** элементов типа **datatype** из массива **inbuf** в массив **outbuf** со сдвигом **position** байт. **outbuf** должен содержать хотя бы **outsize** байт.

MPI

Параметр **`position`** увеличивается на число байт, равное размеру записи. Параметр **`comm`** указывает на коммуникатор, в котором в дальнейшем будет пересылаться сообщение.

Для пересылки упакованных данных используется тип данных **`MPI_PACKED`**.

MPI

```
int MPI_Unpack(void *inbuf, int
insize, int *position, void
*outbuf, int outcount,
MPI_Datatype datatype, MPI_Comm
comm)
```

Распаковка из массива **inbuf** со сдвигом **position** байт в массив **outbuf** **outcount** элементов типа **datatype**.

MPI

```
int MPI_Pack_size(int incount,
MPI_Datatype datatype, MPI_Comm
comm, int *size)
```

Определение необходимого объёма памяти (в байтах) для упаковки **`incount`** элементов типа **`datatype`**.

MPI

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
   int size, rank, position, i;
   float a[10];
   char b[10], buf[100];
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   for(i = 0; i<10; i++){
      a[i] = rank + 1.0;
      if(rank==0) b[i]='a';
      else b[i] = 'b';
   }
   position=0;
```

MPI

```
   if(rank==0){
      MPI_Pack(a, 10, MPI_FLOAT, buf, 100, &position,
MPI_COMM_WORLD);
      MPI_Pack(b, 10, MPI_CHAR, buf, 100, &position,
MPI_COMM_WORLD);
      MPI_Bcast(buf, 100, MPI_PACKED, 0, MPI_COMM_WORLD);
   } else{
      MPI_Bcast(buf, 100, MPI_PACKED, 0, MPI_COMM_WORLD);
      MPI_Unpack(buf, 100, &position, a, 10, MPI_FLOAT,
MPI_COMM_WORLD);
      MPI_Unpack(buf, 100, &position, b, 10, MPI_CHAR,
MPI_COMM_WORLD);
   }
   for(i = 0; i<10; i++) printf("process %d a=%f b=%c\n",
rank, a[i], b[i]);
   MPI_Finalize();}
```

MPI

Односторонние коммуникации

MPI

Применение односторонних коммуникаций позволяет задавать все параметры, относящиеся к пересылке данных, только на стороне посылающего или принимающего процесса. Для этого необходимо создать *окно*, в рамках которого далее возможно осуществлять односторонние коммуникации. При этом, кроме собственно функций посылки данных, появляется необходимость дополнительных операций синхронизации.

MPI

```
int MPI_Win_create(void *base,
MPI_Aint size, int disp_unit,
MPI_Info info, MPI_Comm comm,
MPI_Win *win)
```

Коллективная операция, создающая на всех процессах интракоммуникатора **comm** окно, начинающееся с адреса **base** размером **size** байт (размер окна может равняться **0**). Окно может использоваться для односторонних коммуникаций внутри коммуникатора **comm**.

MPI

`disp_unit` задаёт размер элемента данных для упрощения адресной арифметики: на это значение будут умножаться смещения, задающие расположение данных в окне. Аргумент **`info`** задаёт возможные оптимизации времени выполнения, связанные с использованием окна.

MPI

```
int MPI_Win_free(MPI_Win *win)
```

Коллективная операция, удаляющую окно. Указатель **`win`** устанавливается в значение **`MPI_WIN_NULL`**. Процедура может быть вызвана процессом только после того, как завершено его участие во всех односторонних операциях, связанных с данным окном.

MPI

```
int MPI_Win_get_group(MPI_Win
win, MPI_Group *group)
```

Возвращает в аргументе **`group`** копию группы процессов, соответствующей коммуникатору, для которого создавалось окно **`win`**.

MPI

```
int MPI_Put(void *origin_addr,
int origin_count, MPI_Datatype
origin_datatype, int
target_rank, MPI_Aint
target_disp, int target_count,
MPI_Datatype target_datatype,
MPI_Win win)
```

Односторонняя посылка **origin_count** элементов данных типа **origin_datatype**, начинающихся с адреса **origin_addr**, процессу **target_rank**.

MPI

На процессе **target_rank** данные размещаются со сдвигом **target_disp** от начала окна **win** и интерпретируются как **target_count** элементов типа **target_datatype**. Адрес начала расположения данных вычисляется как

target_addr = window_base + target_disp disp_unit,

где **window_base** и **disp_unit** – адрес начала и размер элемента данных, заданные при создании окна **win**.

MPI

```
int MPI_Get(void *origin_addr,
int origin_count, MPI_Datatype
origin_datatype, int
target_rank, MPI_Aint
target_disp, int target_count,
MPI_Datatype target_datatype,
MPI_Win win)
```

Односторонний приём **origin_count** элементов данных типа **origin_datatype** в массив по адресу **origin_addr** от процесса **target_rank**.

MPI

На процессе **target_rank** данные размещаются со сдвигом **target_disp** от начала окна **win** и интерпретируются как **target_count** элементов типа **target_datatype**. Адрес начала расположения данных вычисляется как

target_addr = window_base + target_disp disp_unit,

где **window_base** и **disp_unit** – адрес начала и размер элемента данных, заданные при создании окна **win**.

MPI

```
int MPI_Accumulate(void
*origin_addr, int origin_count,
MPI_Datatype origin_datatype,
int target_rank, MPI_Aint
target_disp, int target_count,
MPI_Datatype target_datatype,
MPI_Op op, MPI_Win win)
```

Совмещение односторонней посылки данных, аналогичной **MPI_Put**, с выполнением некоторой операции **op**.

MPI

Операция **op** выполняется поэлементно над посылаемыми данными и данными, находящимися в буфере приёма. В качестве **op** могут использоваться все предопределенные операции, допустимые в **MPI_Reduce**. Определяемые пользователем операции не допускаются. В качестве **op** можно использовать константу **MPI_REPLACE**, что означает замещение данных в буфере приёма посылаемыми данными (полный аналог **MPI_Put**).

MPI

Процедуры **`MPI_Put`**, **`MPI_Get`** и **`MPI_Accumulate`** запускаются как неблокирующие. Для того чтобы гарантировать завершение соответствующих операций, предоставляется ряд механизмов синхронизации.

MPI

```
int MPI_Win_fence(int assert,
MPI_Win win)
```

Коллективная операция, выполняющая синхронизацию процессов по доступу к окну **`win`**. Выход из процедуры означает, что все односторонние коммуникации, связанные с окном **`win`**, завершены.

MPI

Аргумент **`assert`** задаётся комбинацией (при помощи побитовой операции «или) признаков, указывающих на возможность некоторых системных оптимизаций. В некоторых случаях их применение может позволить, например, избежать ненужных синхронизаций. Если оптимизаций не требуются, то можно в качестве **`assert`** всегда задавать значение **`0`**.

MPI

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
   int rank, size, prev, next;
   int buf[2];
   MPI_Aint lb, extent;
   MPI_Win win;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   prev = rank - 1;
   next = rank + 1;
```

MPI

```
   if(rank==0) prev = size - 1;
   if(rank==size - 1) next = 0;
   MPI_Type_get_extent(MPI_INT, &lb, &extent);
   MPI_Win_create(buf, 2*extent, extent, MPI_INFO_NULL,
MPI_COMM_WORLD, &win);
   MPI_Win_fence(0, win);
   MPI_Put(&rank, 1, MPI_INT, prev, 1, 1, MPI_INT, win);
   MPI_Put(&rank, 1, MPI_INT, next, 0, 1, MPI_INT, win);
   MPI_Win_fence(0, win);
   MPI_Win_free(&win);
   printf("process %d prev = %d next=%d\n", rank, buf[0],
buf[1]);
   MPI_Finalize();
}
```

MPI

```
int MPI_Win_start(MPI_Group
group, int assert, MPI_Win win)
```

Начало секции, в рамках которой возможны односторонние коммуникации с посылкой данных из вызвавшего процесса в окно **`win`** процессов группы **`group`**. Процессы группы **`group`** должны сделать соответствующий вызов **`MPI_Win_post`**. Если такой вызов не сделан, текущий процесс может быть заблокирован на выполнении операции **`MPI Win start`**.

MPI

```
int MPI_Win_complete(MPI_Win
win)
```

Конец секции, в рамках которой возможны односторонние коммуникации с посылкой данных из вызвавшего процесса в окно **`win`**. Процесс блокируется до тех пор, пока не будут завершены все односторонние коммуникации, инициированные внутри данной секции.

MPI

```
int MPI_Win_post(MPI_Group
group, int assert, MPI_Win win)
```

Начало секции, в рамках которой возможны односторонние коммуникации с процессов группы **`group`** в окно **`win`** вызвавшего процесса. Процессы группы **`group`** должны сделать соответствующий вызов **`MPI_Win_start`**.

MPI

`int MPI_Win_wait(MPI_Win win)`

Конец секции, в рамках которой возможны односторонние коммуникации в окно **`win`** вызвавшего процесса. Процесс блокируется до тех пор, пока все процессы группы **`group`**, заданной в соответствующем вызове **`MPI_Win_post`**, не вызовут процедуру **`MPI_Win_complete`**, что будет означать завершение всех односторонних коммуникаций в окно **`win`** вызвавшего процесса.

MPI

```
int MPI_Win_test(MPI_Win win,
int *flag)
```

Неблокирующая проверка завершённости односторонних коммуникаций в окно **win** вызвавшего процесса. Возвращает в аргументе **flag** значение **1**, если все процессы группы **group**, заданной в соответствующем вызове **MPI_Win_post**, вызвали процедуру **MPI_Win_complete**, и значение **0** иначе. В первом случае действие аналогично процедуре **MPI_Win_wait**.

MPI

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
   int rank, size, prev, next, ranks[2];
   int buf[2];
   MPI_Aint lb, extent;
   MPI_Win win;
   MPI_Group group, commgroup;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   prev = rank - 1;
   next = rank + 1;
```

MPI

```
   if(rank==0) prev = size - 1;
   if(rank==size - 1) next = 0;
   MPI_Type_get_extent(MPI_INT, &lb, &extent);
   MPI_Win_create(buf, 2*extent, extent, MPI_INFO_NULL,
MPI_COMM_WORLD, &win);
   MPI_Comm_group(MPI_COMM_WORLD, &group);
   ranks[0]=prev; ranks[1]=next;
   MPI_Group_incl(group, 2, ranks, &commgroup);
   MPI_Win_post(commgroup, 0, win);
   MPI_Win_start(commgroup, 0, win);
   MPI_Put(&rank, 1, MPI_INT, prev, 1, 1, MPI_INT, win);
   MPI_Put(&rank, 1, MPI_INT, next, 0, 1, MPI_INT, win);
   MPI_Win_complete(win);
   MPI_Win_wait(win);
   MPI_Win_free(&win);
```

MPI

```
    MPI_Group_free(&group);
    MPI_Group_free(&commgroup);
    printf("process %d prev = %d next=%d\n", rank, buf[0],
buf[1]);
    MPI_Finalize();
}
```

MPI

```
int MPI_Win_lock(int lock_type,
int rank, int assert, MPI_Win
win)
```

Синхронизация процессов путём захвата замка. Вызвавший процесс, если необходимо, дожидается освобождения замка, закрывающего окно **`win`** процесса **`rank`**, и захватывает его. После этого возможны односторонние коммуникации с вызвавшего процесса в окно **`win`** процесса **`rank`**.

MPI

Параметр **lock_type** задаёт тип замка:

MPI_LOCK_EXCLUSIVE означает, что одновременно с вызвавшим процессом невозможны односторонние коммуникации других процессов в окно **win** процесса **rank**;

MPI_LOCK_SHARED означает, что одновременно с вызвавшим процессом возможны односторонние коммуникации и других процессов в окно **win** процесса **rank**.

MPI

```
int MPI_Win_unlock(int rank,
MPI_Win win)
```

Освобождение замка, закрывающего окно **`win`** процесса **`rank`**. Вызвавший процесс блокируется, пока не завершатся все односторонние коммуникации с данного процесса в окно **`win`** процесса **`rank`**.

MPI

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
   int rank, size, prev, next;
   int buf[2];
   MPI_Aint lb, extent;
   MPI_Win win;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   prev = rank - 1;
   next = rank + 1;
   if(rank==0) prev = size - 1;
   if(rank==size - 1) next = 0;
```

MPI

```
    MPI_Type_get_extent(MPI_INT, &lb, &extent);
    MPI_Win_create(buf, 2*extent, extent, MPI_INFO_NULL,
MPI_COMM_WORLD, &win);
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, prev, 0, win);
    MPI_Put(&rank, 1, MPI_INT, prev, 1, 1, MPI_INT, win);
    MPI_Win_unlock(prev, win);
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, next, 0, win);
    MPI_Put(&rank, 1, MPI_INT, next, 0, 1, MPI_INT, win);
    MPI_Win_unlock(next, win);
    MPI_Win_free(&win);
    printf("process %d prev = %d next=%d\n", rank, buf[0],
buf[1]);
    MPI_Finalize();
}
```

MPI

Обработка ошибок

MPI

Обработчики ошибок в MPI могут быть привязаны к одному из трёх типов объектов: коммуникатору, окну или файлу. Процедура обработки ошибок будет вызвана при возникновении любого исключения, связанного с таким объектом. Если вызов функции MPI не относится ни к одному такому объекту, то он считается привязанным к коммуникатору **`MPI_COMM_WORLD`**. Обработка ошибок является чисто локальной: каждый процесс может назначить свой обработчик ошибки одному и тому же объекту.

MPI

Предопределённые обработчики ошибок:

MPI_ERRORS_ARE_FATAL - программа будет прервана на всех работающих процессах;

MPI_ERRORS_RETURN не делает ничего, кроме предоставления кода ошибки пользователю.

По умолчанию обработчик **MPI_ERRORS_ARE_FATAL** связывается с коммуникатором **MPI_COMM_WORLD** сразу после его инициализации. Если не предусмотрено иного, любая возникшая ошибка приводит к завершению программы.

MPI

Если пользователь работает с вызовами процедур MPI, анализируя возвращаемые ими коды, он может использовать обработчик ошибок **`MPI_ERRORS_RETURN`**. Однако это не всегда удобно, лучше написать специальный обработчик ошибок.

Использование обработчика ошибок не обязательно позволяет продолжать выполнение программы с вызовами MPI. Целью должна быть выдача сообщений об ошибках и некоторые действия, необходимые перед завершением программы.

MPI

```
int
MPI_Comm_create_errhandler(MPI_C
omm_errhandler_function
*function, MPI_Errhandler
*errhandler)
```

Процедура создаёт связанный с коммуникатором обработчик ошибок **`errhandler`**, реализованный пользовательской процедурой **`function`**.

MPI

На языке Си процедура **function** должна иметь следующий интрефейс:

```
typedef void
MPI_Comm_errhandler_function(MPI
_Comm *, int *, ...);
```

Первый аргумент - коммуникатор, второй – код ошибки, возвращаемый функцией MPI, в которой эта ошибка возникла. Если процедура вернёт **MPI_ERR_IN_STATUS**, это означает, что код ошибки возвратится в поле структуры **status** для запроса, в котором возникла ошибка. Остальные аргументы зависят от реализации.

MPI

```
int
MPI_Comm_set_errhandler(MPI_Comm
comm, MPI_Errhandler errhandler)
```

Процедура связывает с коммуникатором **`comm`** новый обработчик ошибок **`errhandler`**. Обработчиком ошибок может быть либо предопределённый обработчик, либо обработчик, созданный при помощи вызова **`MPI_Comm_create_errhandler`**.

MPI

```
int
MPI_Comm_get_errhandler(MPI_Comm
comm, MPI_Errhandler
*errhandler)
```

Возвращает в аргументе **errhandler** обработчик ошибок, ассоциированный с коммуникатором **comm**. Операция может быть полезна, например, при написании библиотечной процедуры, когда сначала запоминается текущий обработчик, затем присваивается и используется новый, а перед выходом из процедуры восстанавливается первоначальный обработчик.

MPI

```
int
MPI_Win_create_errhandler(MPI_Wi
n_errhandler_function *function,
MPI_Errhandler *errhandler)
```

Процедура создаёт связанный с окном обработчик ошибок **`errhandler`**, реализованный пользовательской процедурой **`function`**.

MPI

На языке Си процедура **function** должна иметь следующий интрефейс:

```
typedef void
MPI_Win_errhandler_function(MPI_
Win *, int *, ...);
```

Первый аргумент - окно, второй – код ошибки, возвращаемый функцией MPI, в которой эта ошибка возникла. Если процедура вернёт **MPI_ERR_IN_STATUS**, это означает, что код ошибки возвратится в поле структуры **status** для запроса, в котором возникла ошибка. Остальные аргументы зависят от реализации.

MPI

```
int
MPI_Win_set_errhandler(MPI_Win
win, MPI_Errhandler errhandler)
```

Процедура связывает с окном **`win`** новый обработчик ошибок **`errhandler`**. Обработчиком ошибок может быть либо предопределённый обработчик, либо обработчик, созданный при помощи вызова **`MPI_Win_create_errhandler`**.

MPI

```
int
MPI_Win_get_errhandler(MPI_Win
win, MPI_Errhandler *errhandler)
```

Возвращает в аргументе **errhandler** обработчик ошибок, ассоциированный с окном **win**. Операция может быть полезна, например, при написании библиотечной процедуры, когда сначала запоминается текущий обработчик, затем присваивается и используется новый, а перед выходом из процедуры восстанавливается первоначальный обработчик.

MPI

```
int
MPI_File_create_errhandler(MPI_F
ile_errhandler_function
*function, MPI_Errhandler
*errhandler)
```

Процедура создаёт связанный с файлом обработчик ошибок **`errhandler`**, реализованный пользовательской процедурой **`function`**.

MPI

На языке Си процедура **function** должна иметь следующий интрефейс:

```
typedef void
MPI_File_errhandler_function(MPI
_File *, int *, ...);
```

Первый аргумент - файл, второй – код ошибки, возвращаемый функцией MPI, в которой эта ошибка возникла. Если процедура вернёт **MPI_ERR_IN_STATUS**, это означает, что код ошибки возвратится в поле структуры **status** для запроса, в котором возникла ошибка. Остальные аргументы зависят от реализации.

MPI

```
int
MPI_File_set_errhandler(MPI_File
file, MPI_Errhandler errhandler)
```

Процедура связывает с файлом **`file`** новый обработчик ошибок **`errhandler`**. Обработчиком ошибок может быть либо предопределённый обработчик, либо обработчик, созданный при помощи вызова **`MPI_File_create_errhandler`**.

MPI

```
int
MPI_File_get_errhandler(MPI_File
file, MPI_Errhandler
*errhandler)
```

Возвращает в аргументе **errhandler** обработчик ошибок, ассоциированный с файлом **file**. Операция может быть полезна, например, при написании библиотечной процедуры, когда сначала запоминается текущий обработчик, затем присваивается и используется новый, а перед выходом из процедуры восстанавливается первоначальный обработчик.

MPI

```
int
MPI_Errhandler_free(MPI_Errhandl
er *errhandler)
```

Процедура помечает обработчик ошибок **errhandler** для удаления. Собственно удаление произойдёт, когда будут удалены все объекты, ассоциированные с этим обработчиком ошибок. После этого значение **errhandler** будет установлено в **MPI_ERRHANDLER_NULL**.

MPI

```
int MPI_Error_string(int
errorcode, char *string, int
*resultlen)
```

Процедура возвращает в аргументе **string** описание ошибки с кодом **errorcode**. Аргумент **string** должен предоставлять буфер размером как минимум **MPI_MAX_ERROR_STRING** символов. В аргументе **resultlen** возвращается реальная длина записанной строки.

MPI

Коды ошибок, возвращаемые процедурами MPI, полностью зависят от реализации (кроме кода **`MPI_SUCCESS`**, всегда равного **`0`**). Это сделано для того, чтобы реализация могла предоставлять максимальную информацию об ошибках посредством вызова **`MPI_Error_string`**. Однако выделено некоторое подмножество кодов ошибок, называемое *классы ошибок*. Классы ошибок фиксированы.

MPI

```
int MPI_Error_class(int
errorcode, int *errorclass)
```

Процедура возвращает в аргументе **`errorclass`** класс ошибки с кодом **`errorcode`**.

MPI

```
int MPI_Add_error_class(int
*errorclass)
```

Добавление нового класса ошибок. В **`errorclass`** вернётся значение для созданного класса. Процедура локальная и может на разных процессах вернуть разные значения.

MPI

```
int MPI_Add_error_code(int
errorclass, int *errorcode)
```

Процедура создаёт новый код ошибки **`errorcode`**, ассоциированный с классом **`errorclass`**.

MPI

```
int MPI_Add_error_string(int
errorcode, char *string)
```

Процедура ассоциирует строку **string** с кодом (классом) ошибки **errorcode**. Строка должна содержать не более **MPI_MAX_ERROR_STRING** символов. Если с кодом **errorcode** уже была ассоциирована некоторая строка, то она будет заменена на новую. Присваивание новой строки стандартному коду ошибки является ошибочным.

MPI

```
int
MPI_Comm_call_errhandler(MPI_Com
m comm, int errorcode)
```

Процедура вызывает обработчик ошибок, связанный с коммуникатором **comm**, с ошибкой **errorcode**. Если используется стандартный обработчик ошибок **MPI_ERRORS_ARE_FATAL**, все процессы коммуникатора **comm** будут остановлены.

MPI

```
int
MPI_Win_call_errhandler(MPI_Win
win, int errorcode)
```

Процедура вызывает обработчик ошибок, связанный с окном **`win`**, с ошибкой **`errorcode`**. Стандартным обработчиком ошибок для окна также является **`MPI_ERRORS_ARE_FATAL`**.

MPI

```
int
MPI_File_call_errhandler(MPI_Fil
e fh, int errorcode)
```

Процедура вызывает обработчик ошибок, связанный с файлом **`file`**, с ошибкой **`errorcode`**. Стандартным обработчиком ошибок для файла является **`MPI_ERRORS_RETURN`**.

MPI

```
#include <stdio.h>
#include "mpi.h"
static int calls = 0;
static int errors = 0;
void err_function(MPI_Comm *comm, int *err, ...)
{
   if(*err == MPI_ERR_OTHER){
      printf("Error MPI_ERR_OTHER\n");
   }
   else{
      errors++;
      printf("Error code %d\n", *err);
   }
   calls++;
}
```

MPI

```
int main(int argc, char **argv)
{
    MPI_Errhandler errhandler;
    MPI_Init(&argc, &argv);
    MPI_Comm_create_errhandler(err_function,
&errhandler);
    MPI_Comm_set_errhandler(MPI_COMM_WORLD,
errhandler);
    MPI_Comm_call_errhandler(MPI_COMM_WORLD,
MPI_ERR_OTHER);
    MPI_Errhandler_free(&errhandler);
    printf("Error handler was called %d times,
with %d errors\n", calls, errors);
    MPI_Finalize();
}
```

MPI

Задание 5: Напишите программу, в которой все процессы приложения пересылают нулевому процессу структуру, состоящую из ранга процесса и названия узла (полученного с помощью вызова процедуры **`MPI_Get_processor_name`**), на котором данный процесс запущен.

MPI

Задание 6: Измерьте латентность и пропускную способность сети при помощи односторонних коммуникаций.

MPI

Задание 7: Напишите свой обработчик ошибок, который записывает возникающие ошибки в специальный лог-файл.