ПРОСТОЙ ПРИМЕР ПАРАЛЛЕЛЬНОЙ ПРОГРАММЫ

MPI – MESSAGE PASSING INTERFACE

Это спецификация для разработчиков и пользователей библиотекой передачи сообщений.

1992 – 1994 - первое появление стандарта MPI-1
1996 – появление стандарта MPI-2

Наиболее известные реализации стандарта MPI:
MVAPICH, MVAPICH2, OpenMPI - smh11.cc.dvo.ru
MPICH2 - mvs17.cc.dvo.ru

КОМПИЛЯЦИЯ MPI-ПРОГРАММ

	MPICH2	MVAPICH2	OpenMPI
C	mpicc	mpicc	mpicc
C++	mpic++	mpicxx	mpic++	mpicxx	mpic++	mpicxx
Fortran	mpif77	mpif90	mpif77	mpif90	mpif77	mpif90

Компилятор MPI – обертка для компиляторов GNU (gcc, g++), которая линкует соответствующую библиотеку MPI.

Если в системе присутствует несколько реализаций MPI, то можно выбрать одну из них командой mpi-selector

```
mpi-selector --list
        mvapich-1.2.0-3635
        mvapich2-1.6
        openmpi-1.4.3
```

АТРИБУТЫ MPI-ПРОГРАММЫ

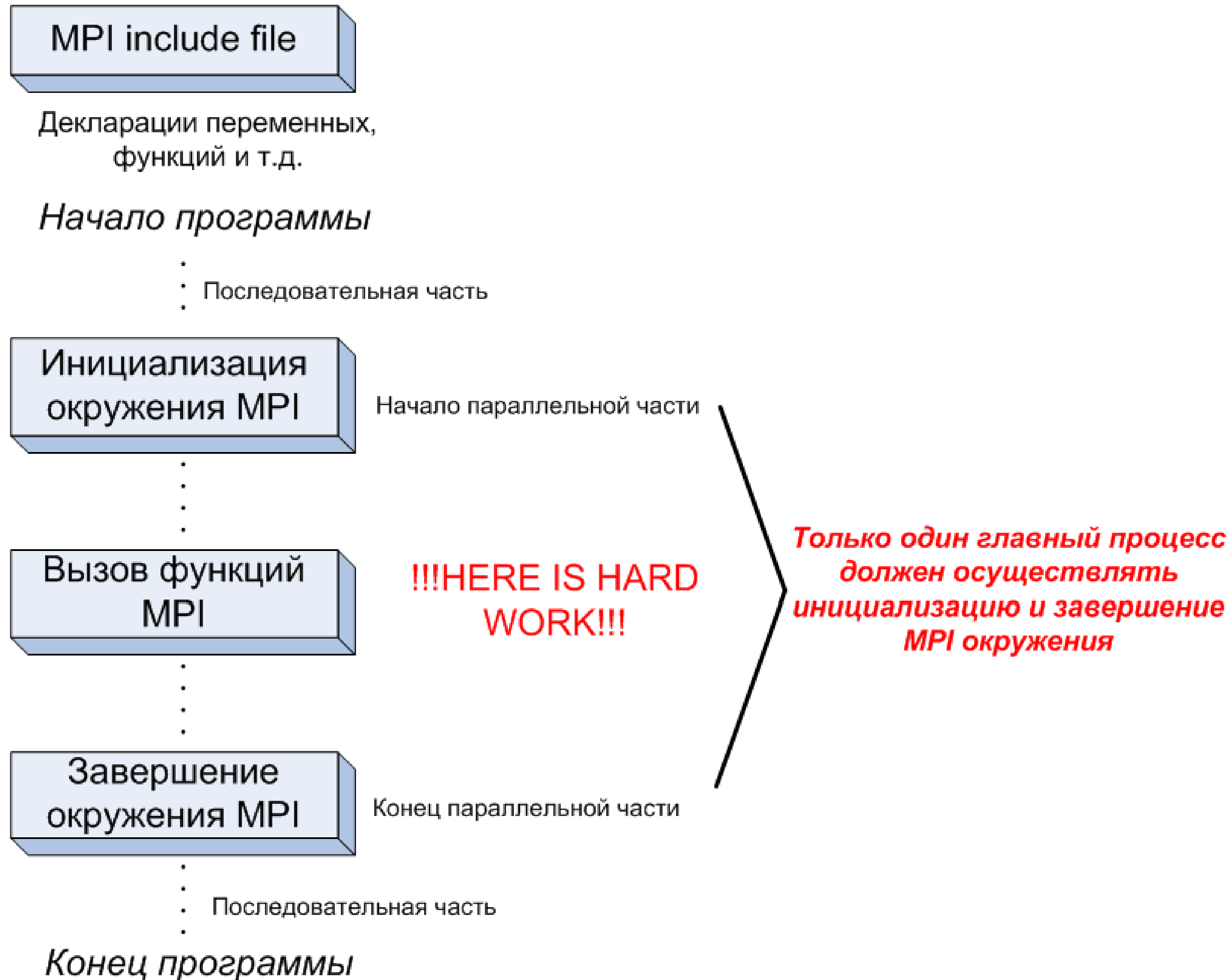
➢Включение заголовочного файла:

C/C++ include file	Fortran include file
#include "mpi.h"	include 'mpif.h'

➢Формат вызова **MPI-**функций:

C/C++	
Формат:	rc = MPI_Xxxxx(parameter, ...);
Пример:	rc = MPI_Bsend(&buf, count, type, dest, tag, comm);
Код ошибки:	Передается в переменную rc. (MPI_SUCCESS)
Fortran	
Формат:	CALL MPI_XXXXX(parameter, ..., ierr) call mpi_xxxxx(parameter, ..., ierr)
Пример:	CALL MPI_BSEND(buf, count, type, dest, tag, comm, ierr)
Код ошибки:	Передается в переменную ierr. (MPI_SUCCESS)

ОБЩАЯ СТРУКТУРА MPI-ПРОГРАММЫ



ПРИМИТИВНЫЕ ТИПЫ ДАННЫХ В MPI

C data type		Fortran data type	
MPI_CHAR	signed char	**MPI_CHARACTER**	character(1)
MPI_SHORT	signed short int		
MPI_INT	signed int	**MPI_INTEGER**	integer
MPI_LONG	signed long int		
MPI_UNSIGNED_CHAR	unsigned char		
MPI_UNSIGNED_SHORT	unsigned short int		
MPI_UNSIGNED	unsigned int		
MPI_FLOAT	float	**MPI_REAL**	real
MPI_DOUBLE	double	**MPI_DOUBLE_PRECISION**	double precision
MPI_LONG_DOUBLE	long double		
		MPI_COMPLEX	complex
		MPI_LOGICAL	Logical
MPI_BYTE	8 binary digits	**MPI_BYTE**	8 binary digits

НЕКОТОРЫЕ ОСНОВНЫЕ ФУНКЦИИ MPI

<table>
<tr><td rowspan="2">MPI_Init</td><td>MPI_Init (&argc,&argv)</td><td rowspan="2">Инициализация среды исполнения MPI-программы. Должна вызываться во ВСЕХ программах MPI и перед всеми другими функциями MPI. Вызывается ТОЛЬКО ОДИН раз.</td></tr>
<tr><td>MPI_INIT (ierr)</td></tr>
<tr><td rowspan="2">MPI_Finalize</td><td>MPI_Finalize ()</td><td rowspan="2">Завершение окружения MPI. Является ПОСЛЕДНЕЙ функцией MPI во ВСЕХ MPI-программах – ни одна MPI-функция не может быть вызвана после нее.</td></tr>
<tr><td>MPI_FINALIZE (ierr)</td></tr>
<tr><td rowspan="2">MPI_Wtime</td><td>MPI_Wtime ()</td><td rowspan="2">Возвращает текущее время в секундах</td></tr>
<tr><td>MPI_WTIME ()</td></tr>
<tr><td rowspan="2">MPI_Comm_rank</td><td>MPI_Comm_rank (comm,&rank)</td><td rowspan="2">Возвращает номер вызывающего процесса (каждый вычислительный процесс внутри своего коммуникатора MPI_COMM_WORLD получает номер из диапазона [0..Ncount-1])</td></tr>
<tr><td>MPI_COMM_RANK (comm, rank, ierr)</td></tr>
<tr><td rowspan="2">MPI_Barrier</td><td>MPI_Barrier (comm)</td><td rowspan="2">Создает барьер синхронизации для группы процессов. Каждый процесс при достижении барьера блокируется пока все процессы из группы не достигнут барьера</td></tr>
<tr><td>MPI_BARRIER (comm, ierr)</td></tr>
</table>

ПРИМЕР №1

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[]){
  int rank, num_proc, rc, tag = 1;
  double t0, t1, time;
  /*MPI SECTION BEGIN*/
  rc = MPI_Init(&argc, &argv);
  MPI_Status Stat;
  if(rc != MPI_SUCCESS){
    printf("\nERROR initializing MPI. Exit.\n");
    MPI_Abort(MPI_COMM_WORLD, rc);
    return 0; }
  t0 = MPI_Wtime();
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
  if( rank == 0) printf("\nBegin example #1\n");
  if( num_proc == 1){
    printf("\nProcess number equals to 1. Exit.\n");
    MPI_Finalize();
    return 0; }
  if( rank != 0) {
    printf("\nHello from process #%d", rank);
    t1 = MPI_Wtime(); time = 1.e6 * (t1 - t0);
    rc = MPI_Send(&time, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);}
  else{
    t1 = MPI_Wtime();
    time = 1.e6 * (t1 - t0);
    FILE *time_file = fopen("./times.txt", "wt");
    for( int i = 1; i < num_proc; i++ ){
      double recv_time;
      rc = MPI_Recv(&recv_time, 1, MPI_DOUBLE, i, tag, MPI_COMM_WORLD, &Stat);
      fprintf(time_file,"#%d process: %9.0f msec\n", i, recv_time);}
    fprintf(time_file,"#%d process: %9.0f msec\n", rank, time);
    fclose(time_file); }
 MPI_Barrier(MPI_COMM_WORLD);
 if( rank == 0 )  printf("\nEnd example\n");
  MPI_Finalize();
  /*MPI SECTION END*/
  return 0;}
```

MPI section

ПРОИЗВОДНЫЕ ТИПЫ ДАННЫХ

➢ Посылка сообщений, которые содержат значения различных типов например, целое число с последующим набором вещественных чисел)
➢Посылка несмежных данных (например, подблоки матрицы)

Производный тип MPI – скрытый объект, который специфицирует две вещи: последовательность базовых типов и последовательность смещений. Последовательность таких пар определяется как *отображение (карта) типа*:
Typemap = $\{(type_0, disp_0), \ldots , (type_{n-1}, disp_{n-1})\}$

Стандартный сценарий определения и использования производных типов:
- Производный тип строится из предопределенных типов MPI и ранее определенных производных типов с помощью специальных функций-конструкторов.
- Новый производный тип регистрируется вызовом функции MPI_Type_commit.
- Уничтожается функцией MPI_Type_free.

Любой тип данных в MPI имеет две характеристики: протяженность и размер, выраженные в байтах:
Протяженность типа определяет, сколько байт переменная данного типа занимает в памяти (опрашивается подпрограммой MPI_Type_extent).
Размер типа определяет количество реально передаваемых байт в коммуникационных операциях (опрашивается подпрограммой MPI_Type_size).

Для простых типов протяженность и размер совпадают.

ПРОИЗВОДНЫЕ ТИПЫ ДАННЫХ -1

C	int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
Fortran	MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
IN count - число элементов базового типа IN oldtype - базовый тип данных OUT newtype - новый производный тип данных	
создает новый тип, элементы которого состоят из указанного числа элементов базового типа, занимающих смежные области памяти	

count = 4

ПРОИЗВОДНЫЕ ТИПЫ ДАННЫХ -2

<table>
<tr><td>C</td><td>int MPI_Type_vector(int count, int blocklength, int stride,MPI_Datatype oldtype, MPI_Datatype *newtype)</td></tr>
<tr><td>Fortran</td><td>MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE,NEWTYPE, IERROR)INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR</td></tr>
<tr><td colspan="2">IN count - число блоков
IN blocklength - число элементов базового типа в каждом блоке
IN stride - шаг между началами соседних блоков, измеренный числом элементов базового типа
IN oldtype - базовый тип данных
OUT newtype - новый производный тип данных</td></tr>
<tr><td colspan="2">создает тип, элемент которого представляет собой несколько равноудаленных друг от друга блоков из одинакового числа смежных элементов базового типа</td></tr>
</table>

oldtype = MPI_REAL

count = 3, blocklenght = 2, stride = 3

newtype

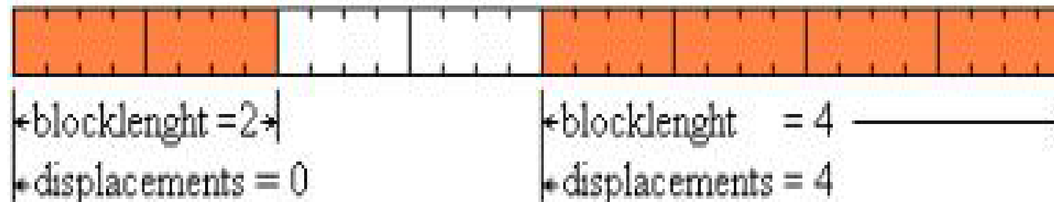
ПРОИЗВОДНЫЕ ТИПЫ ДАННЫХ -3

<table>
<tr><td>C</td><td>int MPI_Type_indexed(int count, int *array_of_blocklengths,int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)</td></tr>
<tr><td>Fortran</td><td>MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),OLDTYPE, NEWTYPE, IERROR</td></tr>
<tr><td colspan="2">IN count - число блоков
IN array_of_blocklengths - массив, содержащий число элементов базового типа в каждом блоке
IN array_of_displacements - массив смещений каждого блока от начала размещения элемента нового типа, смещения измеряются числом элементов базового типа
IN oldtype - базовый тип данных
OUT newtype - новый производный тип данных</td></tr>
<tr><td colspan="2">создает тип, элемент которого состоят из произвольных по длине блоков с произвольным смещением блоков от начала размещения элемента. Смещения измеряются в элементах старого типа.</td></tr>
</table>

ПРОИЗВОДНЫЕ ТИПЫ ДАННЫХ -3

Эта функция создает тип newtype, каждый элемент которого состоит из count блоков, где i-ый блок содержит array_of_blocklengths[i] элементов базового типа и смещен от начала размещения элемента нового типа на array_of_displacements[i] элементов базового типа.

count = 2 blocklenght = (2, 4) displacements = (0,4)



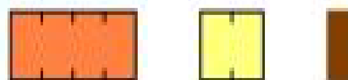
ПРОИЗВОДНЫЕ ТИПЫ ДАННЫХ -4

<table>
<tr><td>C</td><td>int MPI_Type_create_struct(int count, int array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)</td></tr>
<tr><td>Fortran</td><td>MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),ARRAY_OF_TYPES(*), NEWTYPE, IERROR</td></tr>
<tr><td colspan="2">IN count - число блоков
IN array_of_blocklengths - массив, содержащий число элементов одного из базовых типов в каждом блоке
IN array_of_displacements - массив смещений каждого блока от начала размещения структуры, смещения измеряются в байтах
IN array_of_type - массив, содержащий тип элементов в каждом блоке
OUT newtype - новый производный тип данных</td></tr>
<tr><td colspan="2">создает тип, являющийся структурой, состоящей из произвольного числа блоков, каждый из которых может содержать произвольное число элементов одного из базовых типов и может быть смещен на произвольное число байтов от начала размещения структуры.</td></tr>
</table>

ПРОИЗВОДНЫЕ ТИПЫ ДАННЫХ -4

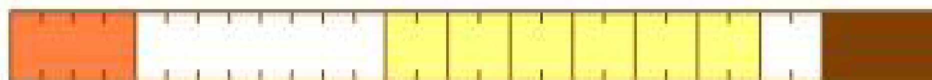
Функция создает тип newtype, элемент которого состоит из count блоков, где i-ый блок содержит array_of_blocklengths[i] элементов типа array_of_types[i]. Смещение i-ого блока от начала размещения элемента нового типа измеряется в байтах и задается в array_of_displacements[i].

oldtypes = (MPI_INT, MPI_SHORT, MPI_CHAR)



count = 3 blocklenght = (1, 6, 4) displacements = (0, 12, 26)

newtype



blocklenght = 1
displacements = 0

blocklenght = 6
displacements = 12

blocklenght = 4
displacements = 26

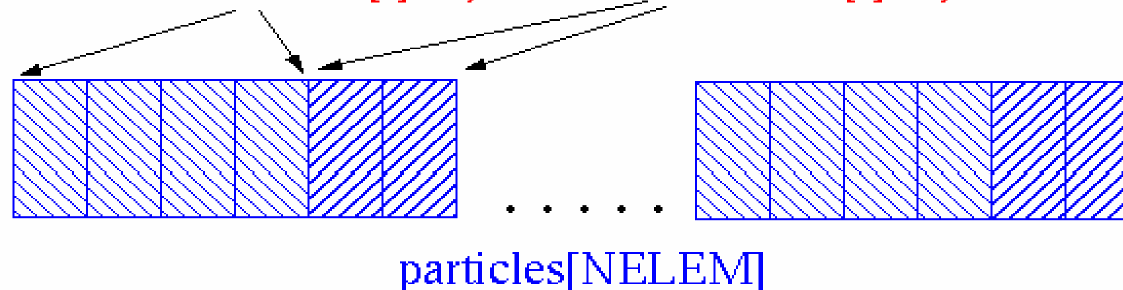
ПРИМЕР ПРОГРАММЫ С ИСПОЛЬЗОВАНИЕМ ПРОИЗВОДНЫХ ТИПОВ ДАННЫХ

MPI_ Type_struct

```
typedef struct { float x, y, z, velocity; int n, type; } Particle;
Particle particles[NELEM];

MPI_Type_extent(MPI_FLOAT, &extent);

count = 2; oldtypes[0] = MPI_FLOAT;     oldtypes[1] = MPI_INT
           offsets[0] = 0;              offsets[1] = 4 * extent;
           blockcounts[0] = 4;          blockcounts[1] = 2;
```




```
MPI_Type_struct(count, blockcounts, offsets, oldtypes, &particletype);

MPI_Send(particles, NELEM, particletype, dest, tag, comm);
```

Sends entire (NELEM) array of particles, each particle being comprised four floats and two integers.

ПОСТАНОВКА ЗАДАЧИ В ОЧЕРЕДЬ MVS17.CC.DVO.RU

TORQUE OpenPBS + MAUI

OpenPBS

qsub {файл паспорта задания}	Запуск задания
qstat	Проверка статуса задания
qdel {номер задачи}	Остановка задачи
qstat –f –Q	Перечень доступных очередей с их описанием
tracejob <JOBID>	Выводит служебную информацию по задаче

MAUI

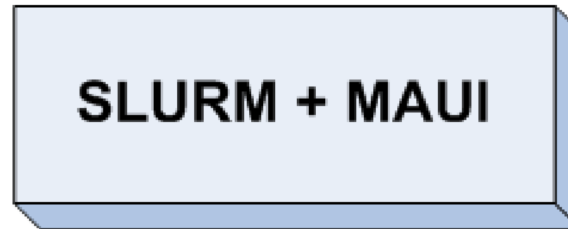
showq	Отражает состояние очереди задач
checkjob <JOBID>	Выводит информацию о задаче
showbf	Показывает количество доступных ресурсов

ПРИМЕР ПАСПОРТА ЗАДАЧИ ДЛЯ OpenBPS

```
#!/bin/bash

#PBS -V
#PBS -S /bin/bash
#PBS -l walltime=00:15:00
#PBS -l nodes=1:ppn=4
#PBS -q simple
#PBS -e /home/daria/Lecture/ex1.err
#PBS -o /home/daria/Lecture/ex1.out
cd $PBS_O_WORKDIR
mpirun -np 8 -machinefile $PBS_NODEFILE ./ex1
```

ПОСТАНОВКА ЗАДАЧИ В ОЧЕРЕДЬ SMH11.CC.DVO.RU



SLURM

srun –n X ExecutableFile [optional params]	Запуск задания
submit –n X ExecutableFile [optional params]	Запуск задания
squeue	Проверка очереди
scancel <JOBID>	Остановка задачи

MAUI

showq	Отражает состояние очереди задач
checkjob <JOBID>	Выводит информацию о задаче
showbf	Показывает количество доступных ресурсов