Системы и средства параллельного программирования

2021 г.

Лектор доцент Н.Н.Попова

Лекция 5
11 октября 2021 г.

Тема

- Понятие модели параллельного программирования
- Этапы разработки параллельных алгоритмов
- Основы модели передачи сообщений.
- MPI – основные понятия, состав
- Методы организации 2-ух точечных обменов в MPI

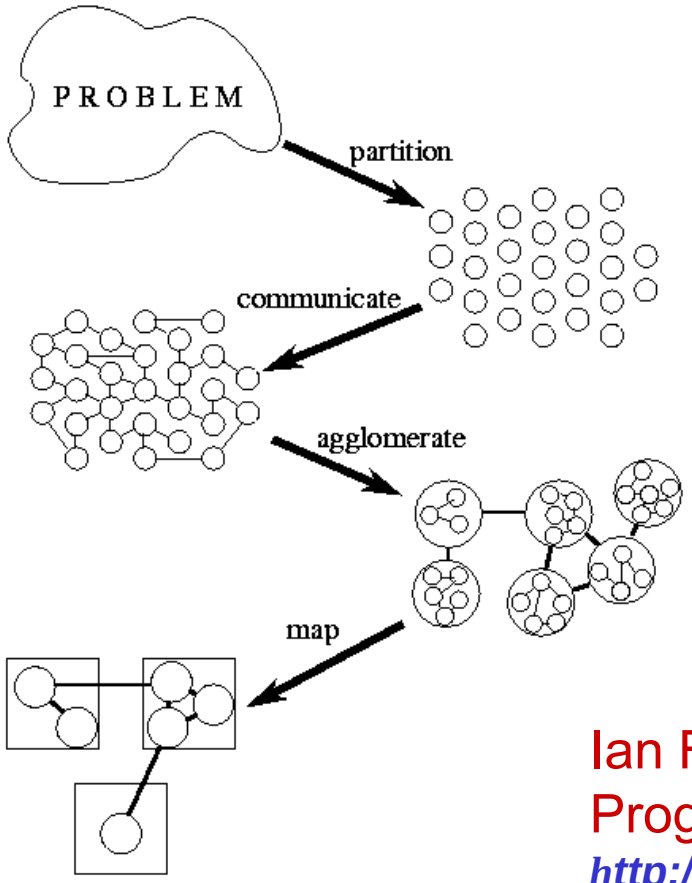
Модель параллельных вычислений

Параллельная вычислительная модель – множество взаимосвязанных механизмов, обеспечивающих следующие требования к организации параллельных вычислений:

– передачу сообщений (**communication)**
- синхронизацию (**synchronization)**
– разделение работ (partitioning)
– размещение работ по процессорам/ядрам (placement)
– управление выполнением работ (scheduling)

Параллельная программа – программа, в которой явно определено параллельное выполнение всей программы либо ее фрагментов (блоков, операторов, инструкций). Программу, в которой параллелизм поддерживается ***неявно***, не будем относить к параллельным.

Этапы разработки параллельных программ



1. Декомпозиция
2. Проектирование коммуникаций
3. Укрупнение
4. Планирование вычислений

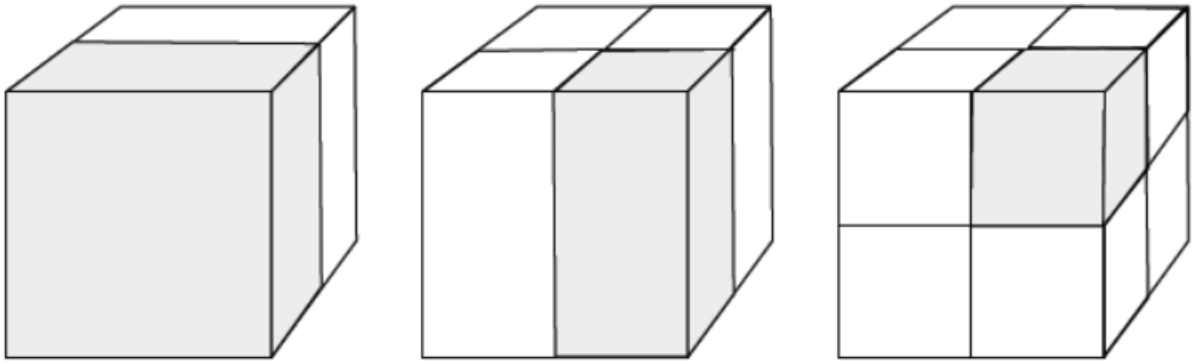
Ian Foster “Designing and Building Parallel Program”

http://www.mcs.anl.gov/~itf/dbpp/text/node4.html

Стратегии декомпозиции

- Domain decomposition : разделение геометрической области на подобласти
- Functional decomposition : разделение алгоритма на несколько компонент
- Independent tasks : разделение вычислений на несколько независимых задач (embarrassingly parallel)
- Array parallelism : одновременное выполнение операций над элементами массивов (векторов, матриц и др.)
- Divide-and-conquer : рекурсивное разделение решаемой задачи на подзадачи с деревоподобной иерархией
- Pipelining : разделение задачи на последовательность этапов

Пример реализации стратегии декомпозиции данных (DD)



1D разбиение
slab

2D разбиение
pencil

3D разбиение
block

Желательные свойства коммуникаций

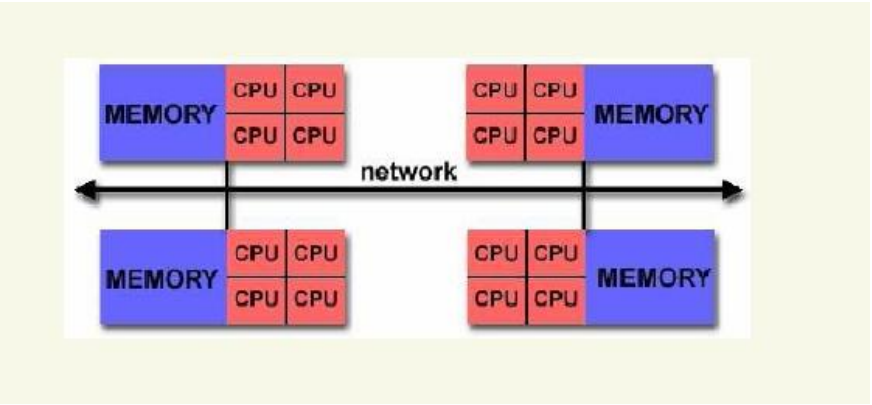
- Минимизация отношения частоты к объему пересылок
- Максимальная локализация (между соседними задачами)
- Равномерное использование ресурсов коммуникационных каналов
- Сохранение параллельности задач
- Максимально возможное совмещение вычислений с передачами

Проблемы мэппинга задач

- Мэппинг должен максимизировать параллельность выполнения задач, минимизировать коммуникации, поддерживать балансировку загрузки и т.д.
- Коммуникации между задачами могут не соответствовать физической топологии коммуникационной сети вычислительной системы
- Две взаимодействующие задачи могут быть назначены на один процессор, сокращая коммуникационные издержки, но сохраняя параллельность
- В общем случае, нахождение оптимального решения является NP-полной задачей, нужны эвристики для ее решения

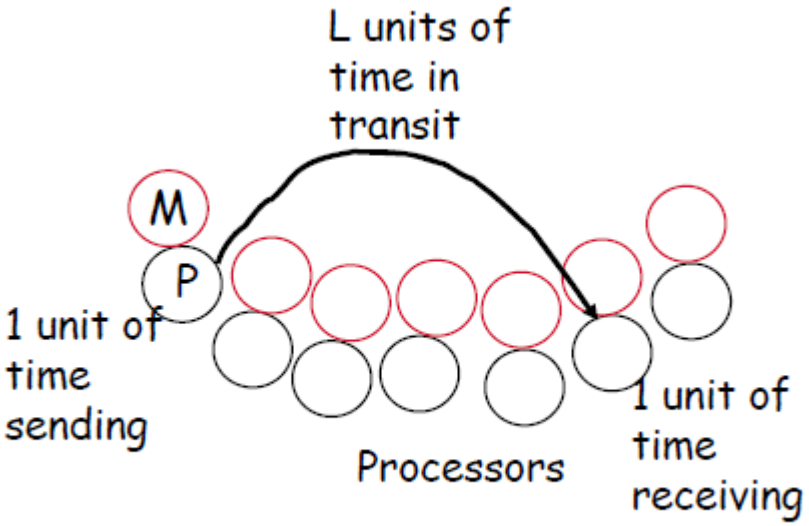
Модели параллельных программ. Системы с распределенной памятью

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



Message Passing Model

Почтовая модель передачи сообщений (1992)



Пример MPI-программы

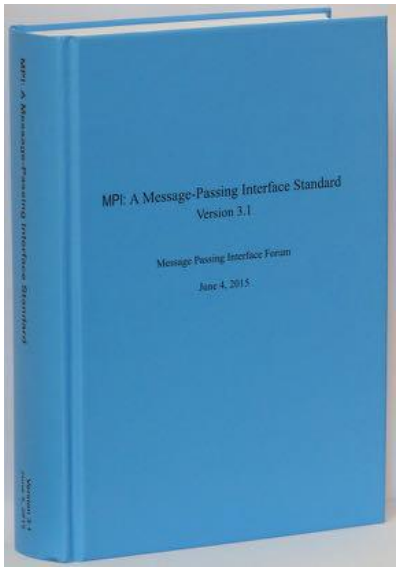
```
#include <stdio.h>
#include <mpi.h>
#define N 1024
int main(int argc, char *argv[])
{ double sum, all_sum;
  double a[N];
  int  i, n =N;
  int size, rank;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,
      &rank);
MPI_Comm_size(MPI_COMM_WORLD,
      &size);
```

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

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
- Стандарты
  - **http://www.mcs.anl.gov/mpi**
  - **http://www.mpi-forum.org/docs/docs.html**

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



Реализации MPI - библиотеки

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

Отличия в реализациях библиотек MPI

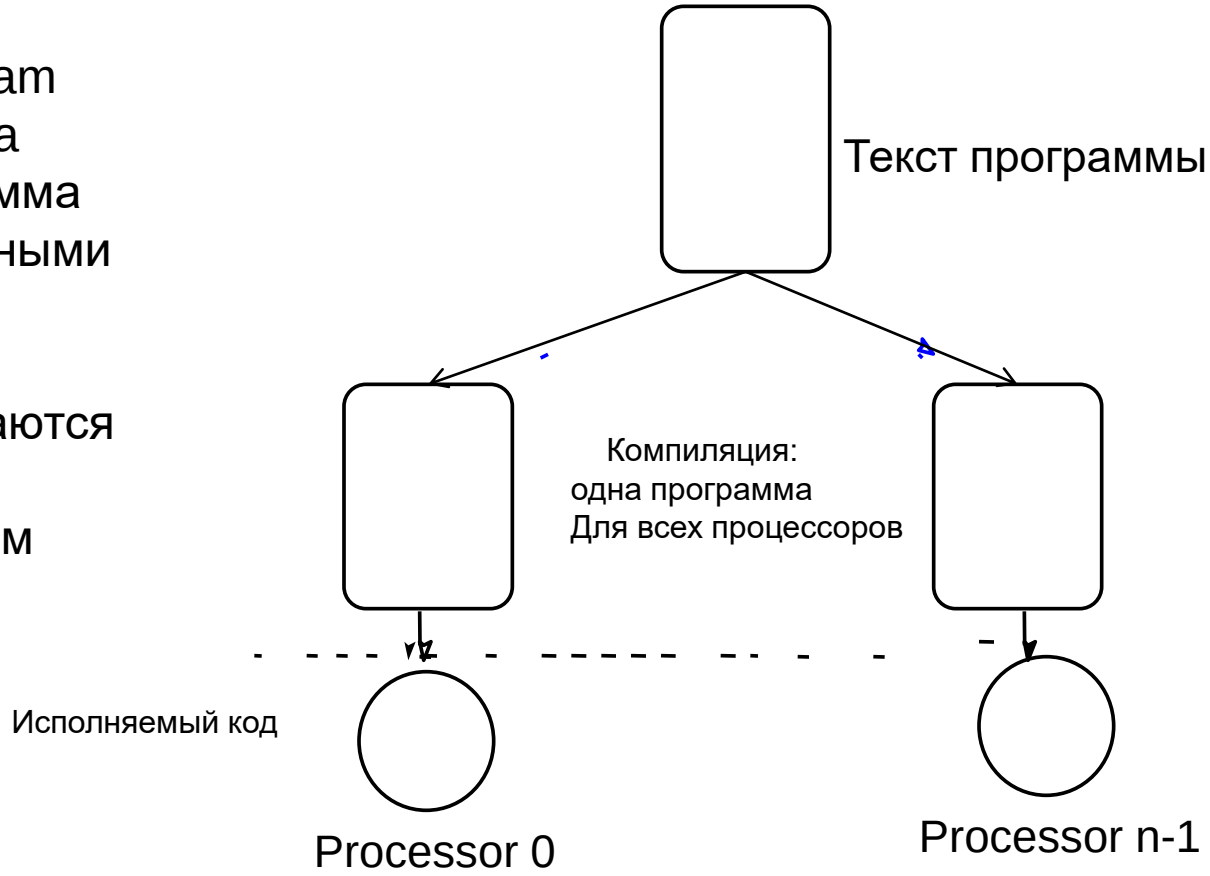
- поддерживаемые вычислительные системы и коммуникационные сети (InfiniBand, Myrinet, SCI и др.)
- реализованные протоколы дифференцированных обменов
- используемые алгоритмы коллективных обменов информацией
- поддерживаемые алгоритмы вложения графов программ в структуры вычислительных систем
- возможность выполнения MPI-функций в многопоточной среде
- полнота реализации стандарта

Модель MPI

- Параллельная программа состоит из процессов, процессы могут быть многопоточными.
- MPI реализует передачу сообщений между процессами.
- Основная схема взаимодействия между 2-мя процессами: схема «рукопожатия» – процессы согласовывают передачу .
- Межпроцессное взаимодействие предполагает:
  - синхронизацию
  - перемещение данных из адресного пространства одного процесса в адресное пространство другого процесса.

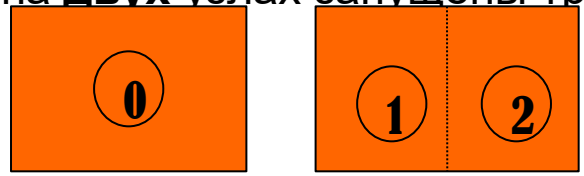
Модель MPI-программ

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



Модель выполнения MPI- программы

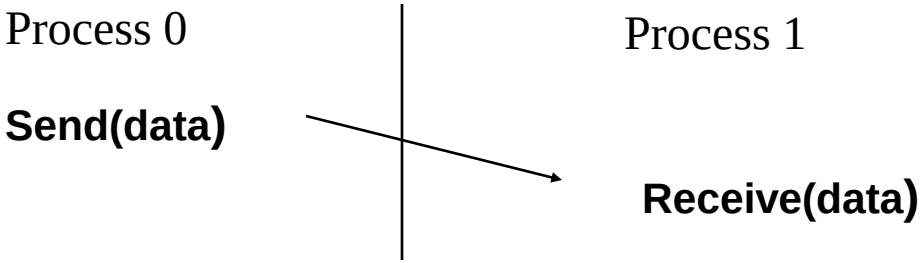
- Запуск: ***mpirun***
- При запуске указываем число требуемых процессоров **np** и название программы: пример: ***mpirun –np 3 prog***
- На выделенных узлах запускается ***np*** копий (процессов) указанной программы
  - Например, на **двух** узлах запущены три копии программы.



- Каждый процесс MPI-программы получает два значения:
  - ***np –*** *число процессов*
  - ***rank*** из диапазона [0 … ***np-1***] – номер процесса
- Любые два процесса могут непосредственно обмениваться данными с помощью функций передачи сообщений

Основы передачи данных в MPI

- Необходимы уточнения процесса передачи

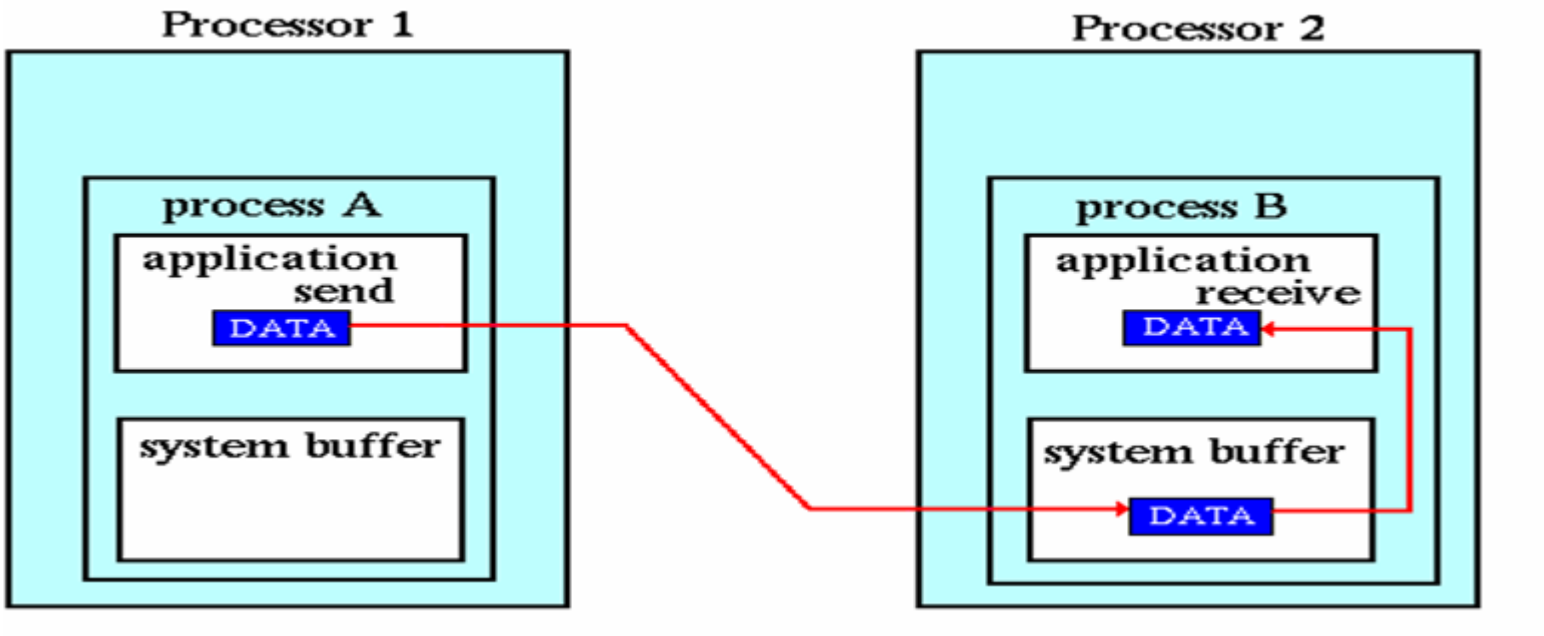


- Требуется уточнить:
  - Как должны быть описаны данные ?
  - Как должны идентифицироваться процессы?
  - Как получатель получит информацию о сообщении?
  - Что значить завершение передачи?

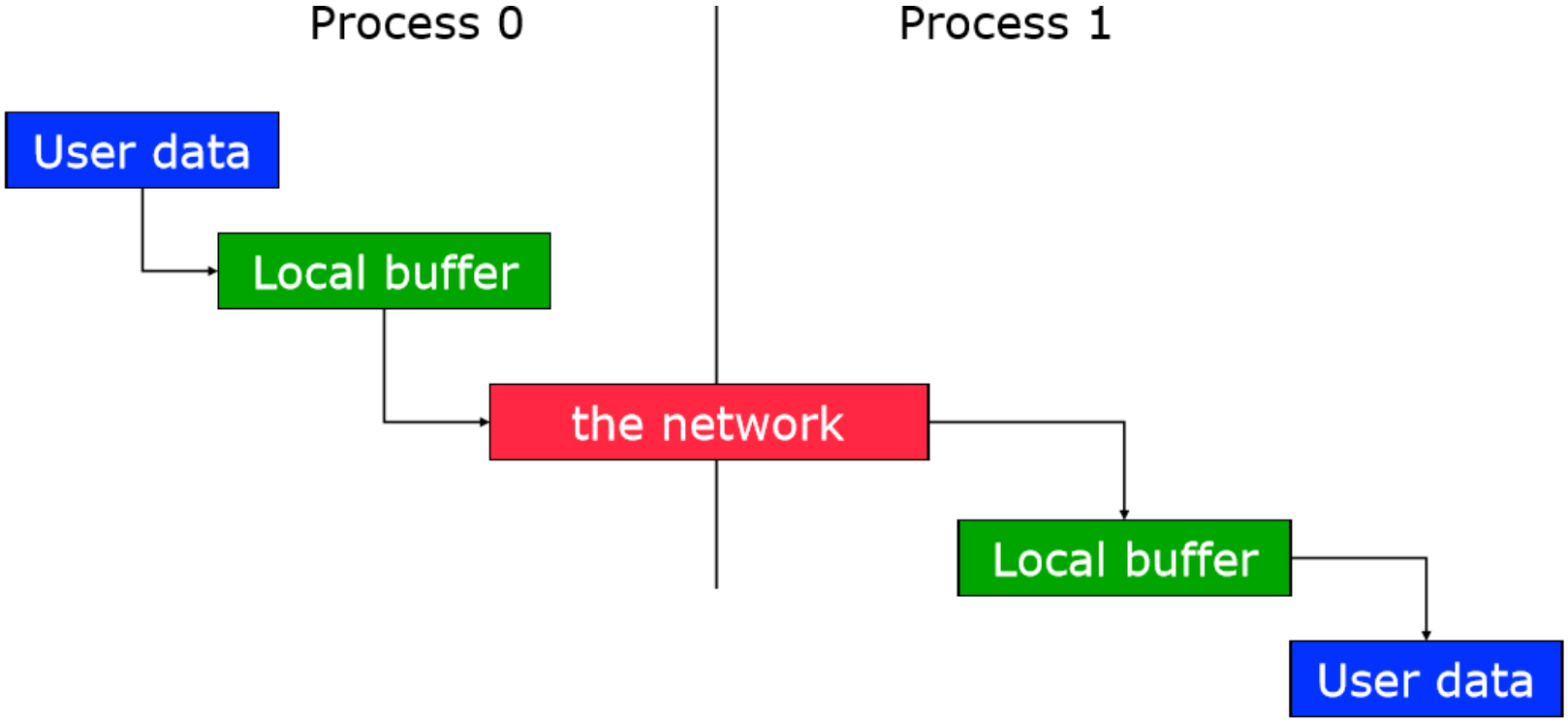
Системный буфер

- Внешний объект по отношению к MPI-программе
- Не оговаривается в стандарте => зависит от реализации
- Имеет конечный размер => может переполняться
- Часто не документируется
- Может существовать как на передающей стороне, так и на принимающей или на обеих сторонах
- Повышает производительность параллельной программы

Схема выполнения операций передачи сообщений



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

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

Базовые MPI-типы данных

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Специальные типы MPI

- MPI_Comm
- MPI_Status
- MPI_datatype

Понятие тэга

- 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.

C: MPI helloworld.c

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv){
    MPI_Init(&argc, &argv);
    printf("Hello, MPI world\n");
    MPI_Finalize();
    return 0;
 }
```

Формат MPI-функций

C (case sensitive):

```
error = MPI_Xxxxx(parameter,...);
MPI_Xxxxx(parameter,...);
```

C++ (case sensitive):

```
error = MPI::Xxxxx(parameter,...);
MPI::Xxxxx(parameter,...);
```

Основные группы функций MPI

- Определение среды
- Передачи «точка-точка»
- Коллективные операции
- Производные типы данных
- Группы процессов
- Виртуальные топологии
- Односторонние передачи данных
- Параллельный ввод-вывод
- Динамическое создание процессов
- Средства профилирования

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 быть первым вызовом функций MPI, 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.
```

http://www.mcs.anl.gov/research/projects/mpi/www/www3/MPI_Init.html

Обработка ошибок MPI-функций

Определяется константой ***MPI_SUCCESS***

Для обработки ошибок необходимо выполнить:

MPI_Comm_set_errhandler (MPI_COMM_WORLD, MPI_ERRORS_RETURN)

MPI_Comm_set_errhandler(MPI_COMM_WORLD,MPI_ERRORS_ARE_FATAL)

```
int error;
……
MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
error = MPI_Init(&argc, &argv));
If (error != MPI_SUCCESS)
{
 fprintf (stderr, “ MPI_Init error \n”);
return 1; /* exit(1);  */
}
```

Количество процессов в коммуникаторе

- Размер коммуникатора

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

Результат – число процессов в коммуникаторе

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Comm_size.html

Вызов MPI-функции до MPI_Init

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
int major,minor;
printf("Compile-time MPI version is %d.%d\n",MPI_VERSION,
MPI_SUBVERSION);
MPI_Get_version(&major,&minor);
printf("Run-time MPI version is %d.%d\n",major,minor);
return 0;
}
```

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

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.

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

Компиляция MPI-программ

- Компиляция

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

  Например:

  mpicc –o hw helloworld.c

- Запуск в интерактивном режиме

  mpirun –np 128 hw

  np – количество MPI-процессов

Компиляция MPI-программ на Polus

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

  mpixlc –o hw helloworld.c
- *mpisubmit.pl [параметры скрипта] исполняемый_файл [-- параметры исполняемого файла]*

  mpisubmit.pl –n 32 hw
- *bjobs* или *bjobs –u all*

Запуск параллельных программ на Polus (http://hpc.cs.msu.su/node/243)

Аргумент	Значение по умолчанию	Описание
-n | --nproc	1	Запрашиваемое число вычислительных узлов
-w | --wtime	00:15	Максимальное время выполнения задания
-t	1	Выставляет переменную окружения OMP_NUM_THREADS равной числу запрашиваемых нитей
--gpu	нет	Указывает, что задачу надо запустить задачу на графических картах
--stdout	<exec>.$(jobid).out	Файл, в который будет направлен стандартный поток вывода
--stderr	<exec>.$(jobid).err	Файл, в который будет направлен стандартный поток ошибок
--stdin		Файл, содержимое которого будет использовано в качестве стандартного ввода
-h | --help		Вывести справочную информацию о параметрах командной строки
-d | --debug		Не ставить задачу в очередь, но вывести содержимое автоматически генерируемого командного файла в stdout.

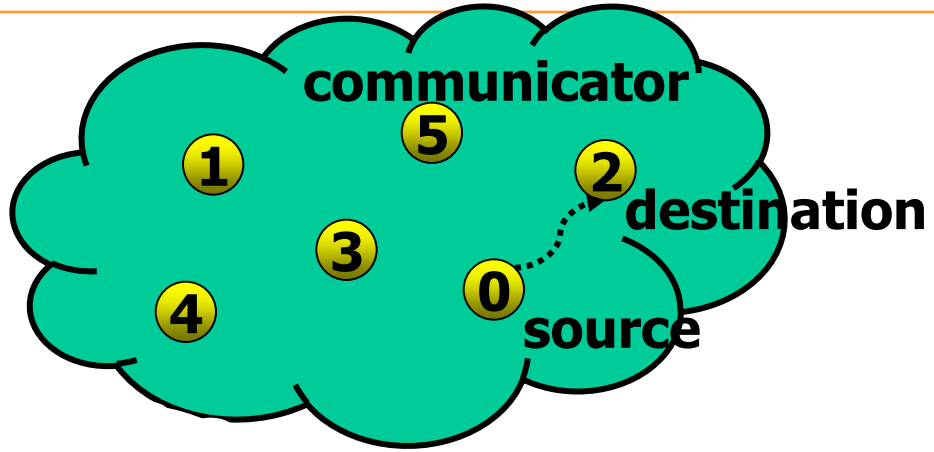
Взаимодействие «точка-точка»

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

Функции MPI передачи «точка-точка»

Point-to-Point Communication Routines		
MPI_Bsend	MPI_Bsend_init	MPI_Buffer_attach
MPI_Buffer_detach	MPI_Cancel	MPI_Get_count
MPI_Get_elements	MPI_Ibsend	MPI_Iprobe
MPI_Irecv	MPI_Irsend	MPI_Isend
MPI_Issend	MPI_Probe	MPI_Recv
MPI_Recv_init	MPI_Request_free	MPI_Rsend
MPI_Rsend_init	MPI_Send	MPI_Send_init
MPI_Sendrecv	MPI_Sendrecv_replace	MPI_Ssend
MPI_Ssend_init	MPI_Start	MPI_Startall
MPI_Test	MPI_Test_cancelled	MPI_Testall
MPI_Testany	MPI_Testsome	MPI_Wait
MPI_Waitall	MPI_Waitany	MPI_Waitsome

Передача сообщений типа «точка-точка»



- Взаимодействие между двумя процессами
- Процесс-отправитель(Source process) ***посылает*** сообщение процессу-получателю (Destination process)
- Процесс-получатель ***принимает*** сообщение
- Передача сообщения происходит в рамках заданного коммуникатора
- Процесс-получатель определяется рангом в коммуникаторе

Завершение передачи

- “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

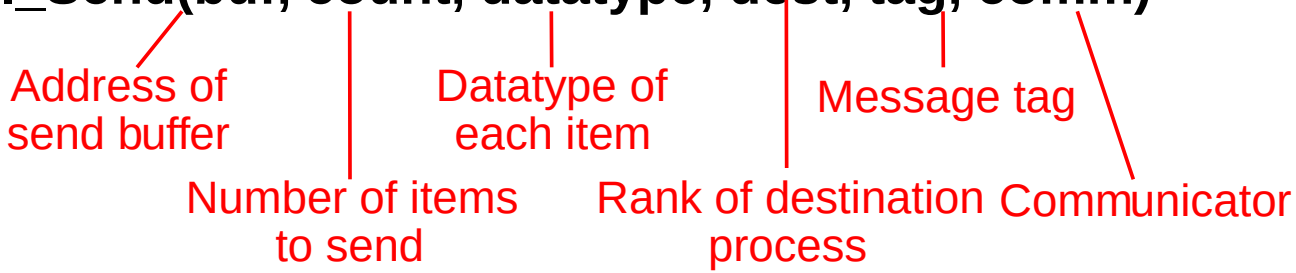
MPI Send

Обобщенная форма:

MPI_SEND (buf, count, datatype, dest, tag, comm)

- Буфер сообщения описывается как (**start, count, datatype**).
- Процесс получатель (**dest)** задается номером (rank) в заданном коммуникаторе (**comm)** .
- По завершению функции буфер может быть использован.

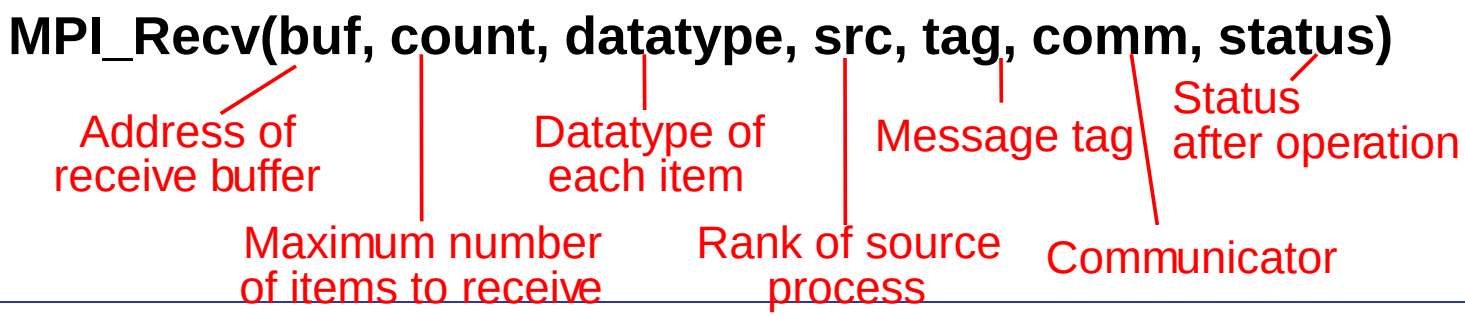
MPI_Send(buf, count, datatype, dest, tag, comm)



MPI Receive

MPI_RECV(buf, count, datatype, source, tag, comm, status)

- Ожидает, пока не придет соответствующее сообщение с заданными **`source`** и **`tag`**
- **`source`** – **номер процесса в коммуникаторе** **`comm`** или **`MPI_ANY_SOURCE`**.
- **`status`** содержит дополнительную информацию



Функция 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)
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Send.html

Функция 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)
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Recv.html

Wildcarding (джокеры)

- Получатель может использовать 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

Информация о завершившемся приеме сообщения

- Возвращается функцией **`MPI_Recv`** через параметр **`status`**

- typedef struct _MPI_Status {
  int count;
  int cancelled;
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR; } MPI_Status, *PMPI_Status;

2 предопределенных **MPI_Status** уазателя, которые могут использоваться:**MPI_STATUS_IGNORE** и **MPI_STATUSES_IGNORE**.

Полученное сообщение

- Может быть меньшего размера, чем указано в функции MPI_Recv
- `count` – число реально полученных элементов можно узнать

с использованием функции:

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

Пример

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv (..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

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
- Выделяется интервал в программе

```
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);
 for (i=0; i<n; i+=) a[i]=size*myrank + i; //array initialization
```

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 */
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); }
```

Вычисление числа π с использованием MPI (1)

```
#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 (2)

```
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 (3)

```
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