

Методы и средства параллельного программирования

2016 г.

Лектор доцент Н.Н.Попова

Лекция 3
3 октября 2016 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 – это множество взаимосвязанных механизмов, обеспечивающих следующие требования к организации параллельных вычислений:

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

Вычислительная модель определяется на различных **уровнях абстракции**:

- Уровне аппаратной организации ВС – например, shared-memory
- Уровне языка программирования
- Уровне алгоритма – например, PRAM (Parallel Random Access Machine), CREW (concurrent read, exclusive write) shared-memory алгоритм

Параллельный алгоритм

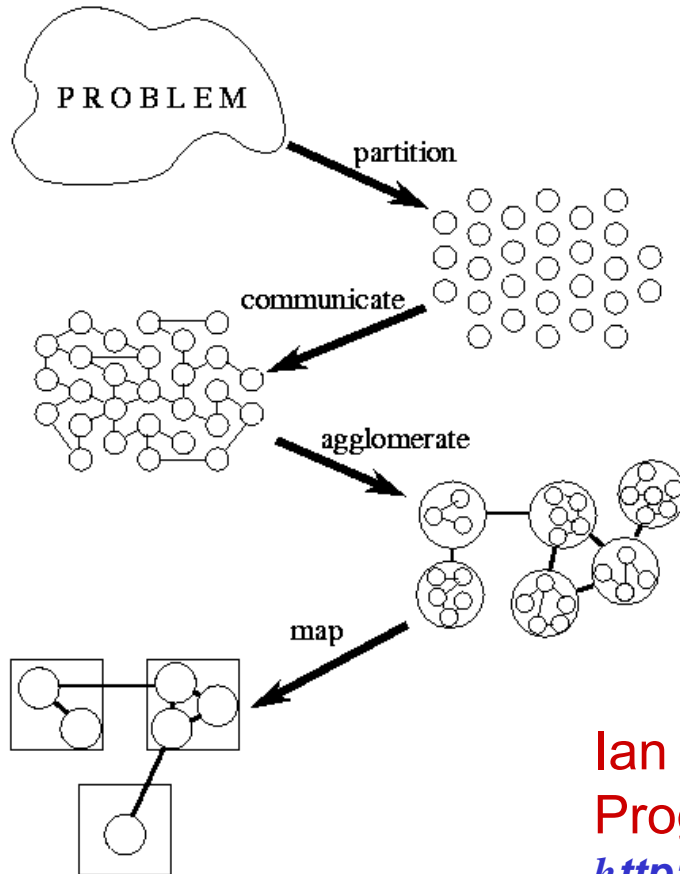
Википедия:

Параллельный алгоритм – алгоритм, который может быть реализован по частям на множестве различных вычислительных устройств с последующим объединением полученных результатов и получением корректного результата.

Параллельная программа

- **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.
- **Процесс** – программа во время выполнения (интуитивно). Существует несколько более формальных определений
Параллельная программа, как правило, выполняется в рамках **нескольких процессов, ВЗАИМОДЕЙСТВУЩИХ!**
- **Поток** – легковесный процесс. В рамках одного процесса может существовать **НЕСКОЛЬКО ПОТОКОВ** – модель OpenMP- программ.

Этапы разработки параллельных программ



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

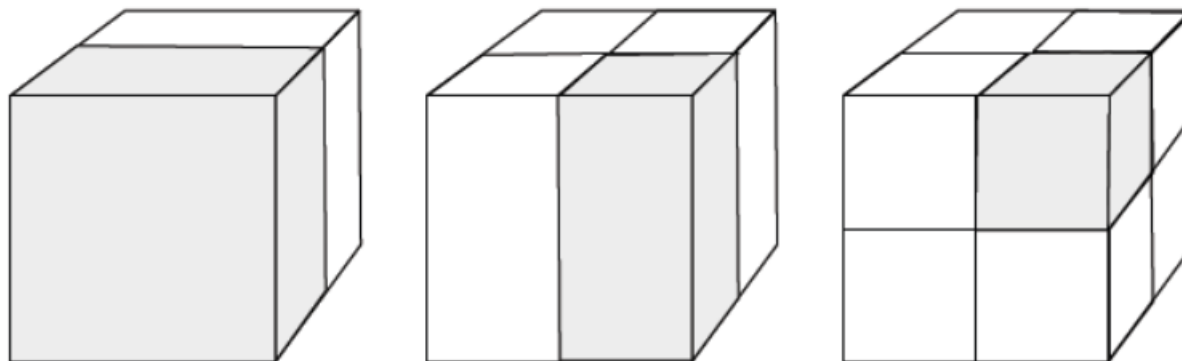
Ian Foster "Designing and Building Parallel Program"

<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 дерево-подобной иерархией
- Pipelining : разделение задачи на последовательность этапов

Пример реализации стратегии декомпозиции данных (DD)



Коммуникационные шаблоны

- Коммуникационные шаблоны определяются зависимостью по данным между задачами вследствие ограниченности локальной памяти
- Коммуникационные шаблоны могут быть:
 - локальными или глобальными
 - структурными или случайными
 - постоянными или динамически изменяемыми
 - синхронными и асинхронными

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

Эффект отношения «Площадь/Объем»

- Для DD вычисления пропорциональны объему подобласти, коммуникации – площади подобласти
- Декомпозиции более высоких размерностей имеют предпочтительные отношения «площадь/объем»
- Разделение по большому числу размерностей ведет к большому числу соседних подобластей, но меньшему объему коммуникаций

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

Возможные стратегии мэппинга

- block mapping : блок последовательных задач назначается на последовательные процессоры
- cyclic mapping : задача i назначается на процессор $i \bmod p$
- reflection mapping : как циклическое, но задачи назначаются в обратном порядке
- block-cyclic mapping and block-reflection mapping : блоки задач назначаются на процессоры как в циклическом мэппинге
- Для многомерных задач этот мэппинг может применяться к каждому из измерений

Модели параллельных программ: аппаратный уровень абстракции

■ Системы с общей памятью

- Программирование, основанное на потоках
- Программа строится на базе последовательной программы
- Возможно автоматическое распараллеливание компилятором с использованием соответствующего ключа компилятора
- Директивы компиляторов (OpenMP, ...)

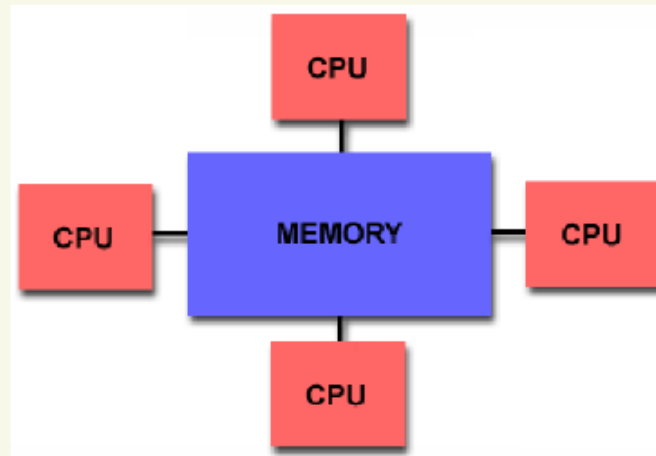
■ Системы с распределенной памятью

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

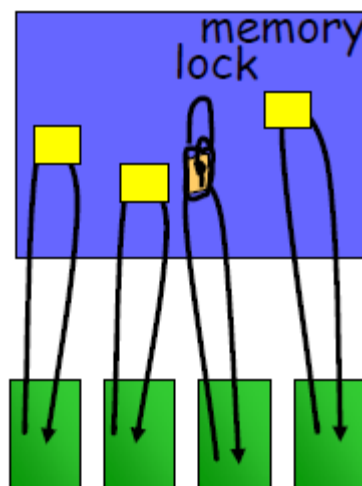
Реализация - Message Passing библиотек:

- MPI (“Message Passing Interface”)
- PVM (“Parallel Virtual Machine”)
- Shmem,

Системы с общей памятью.



Shared Memory Parallel Programming Model

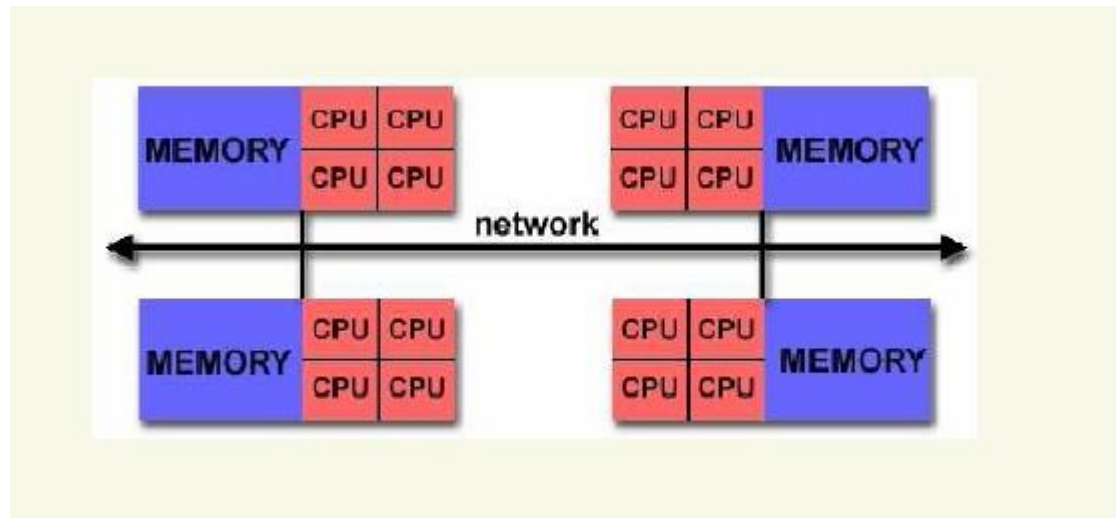


Потоки

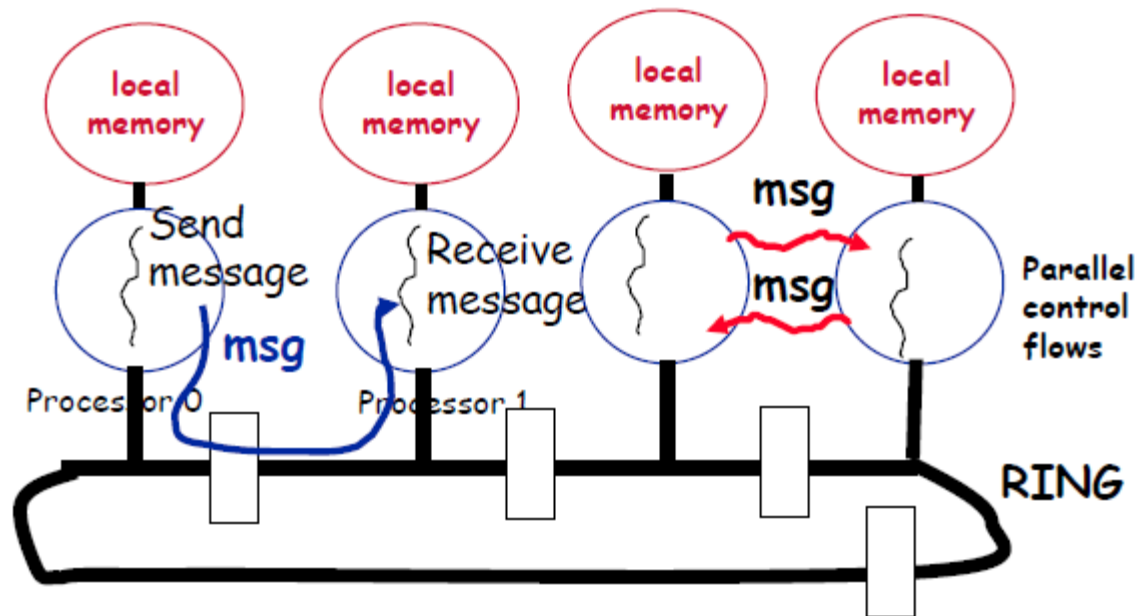
Communication: через общую память

Synchronization: shared memory locks

Системы с распределенной памятью.



Message Passing Parallel Machine Model

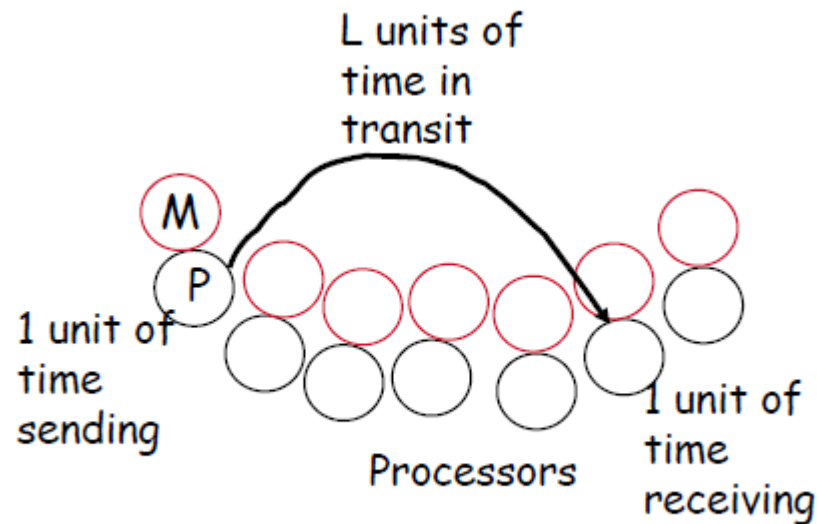


Communication: через сообщения

Synchronization: через сообщения

Message Passing Algorithm Model

Почтовая модель передачи сообщений (1992)



Пример параллельной программы программы (C, OpenMP)

Сумма элементов массива

```
#include <stdio.h>
#define N 1024
int main()
{ double sum;
  double a[N];
  int status, i, n =N;
  for (i=0; i<n; i++){
    a[i] = i*0.5; }
  sum =0;
  #pragma omp for reduction (+:sum)
  for (i=0; i<n; i++)
    sum = sum+a[i];
  printf ("Sum=%f\n", sum);
}
```

MPI

```
#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, myrank;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,
    &myrank);
  MPI_Comm_size(MPI_COMM_WORLD,
    &size);
```

```
n= n/ size;
  for (i=myrank*n; i<n; i++){
    a[i] = i*0.5; }

  sum =0;
  for (i=myrank*n; i<n; i++)
    sum = sum+a[i];
  MPI_Reduce(& sum,& all_sum, 1,
    MPI_DOUBLE, MPI_SUM, 0,
    MPI_COMM_WORLD);
  if ( !myrank)
    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

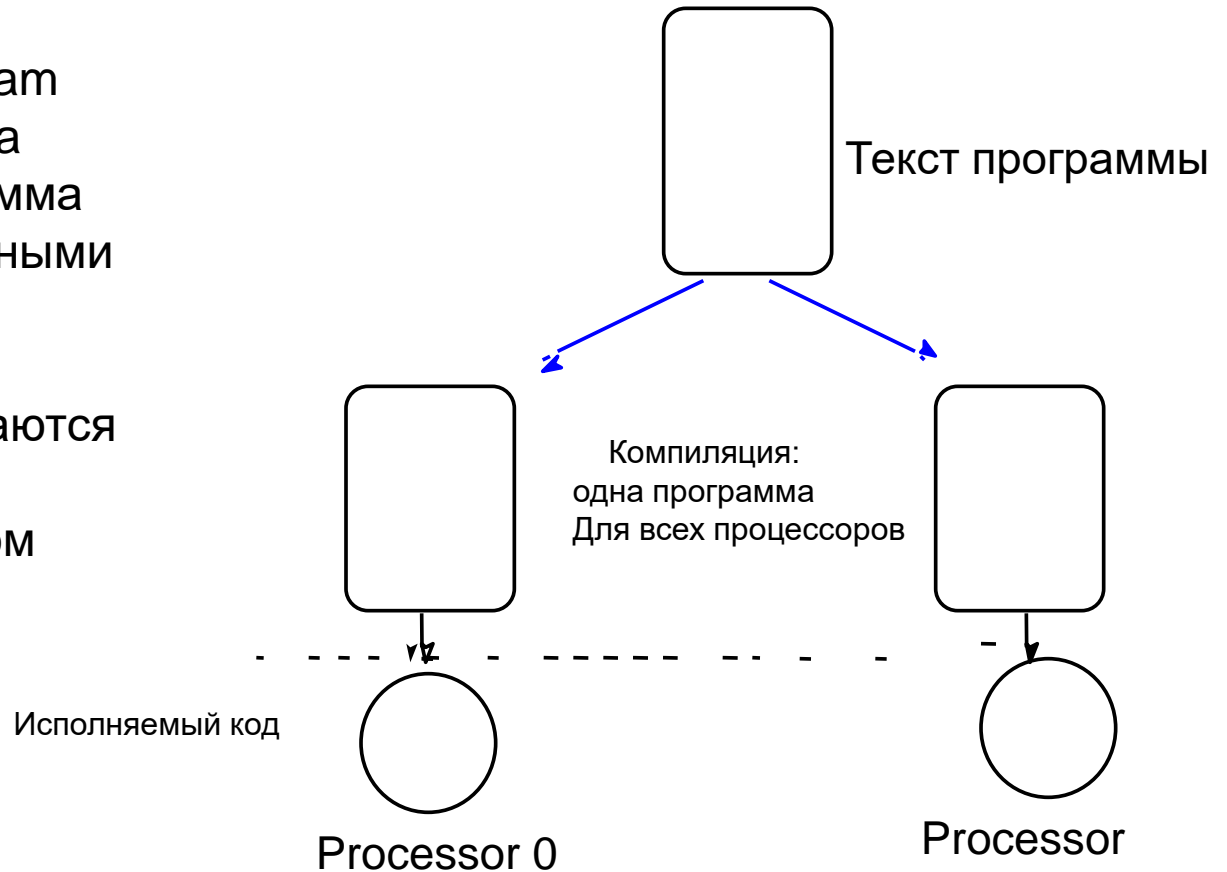
- MPICH
- LAM/MPI
- Mvarich
- OpenMPI
- Коммерческие реализации Intel, IBM и др.

Модель MPI

- Параллельная программа состоит из процессов, процессы могут быть многопоточными.
- MPI реализует передачу сообщений между процессами.
- Межпроцессное взаимодействие предполагает:
 - синхронизацию
 - перемещение данных из адресного пространства одного процесса в адресное пространство другого процесса.

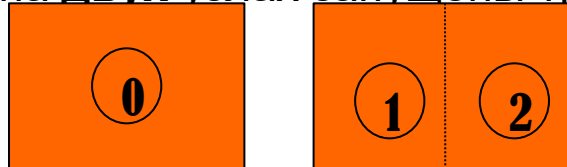
Модель MPI-программ

- **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

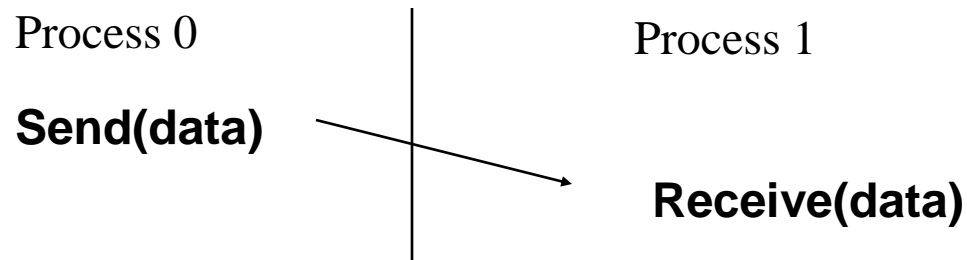
- Запуск: *mpirun*
- При запуске указываем число требуемых процессоров *np* и название программы: пример: *mpirun -np 3 prog*
- На выделенных узлах запускается *np* копий (процессов) указанной программы
 - Например, на ~~двух~~ узлах запущены три копии программы.



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

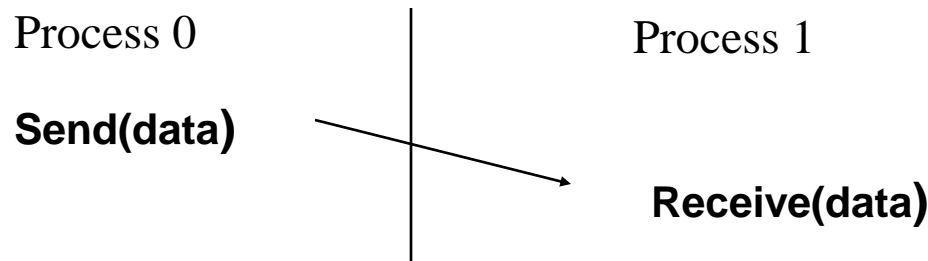
Основы передачи данных в MPI

- Данные посылаются одним процессом и принимаются другим.
- Передача и синхронизация совмещены.



Основы передачи данных в MPI

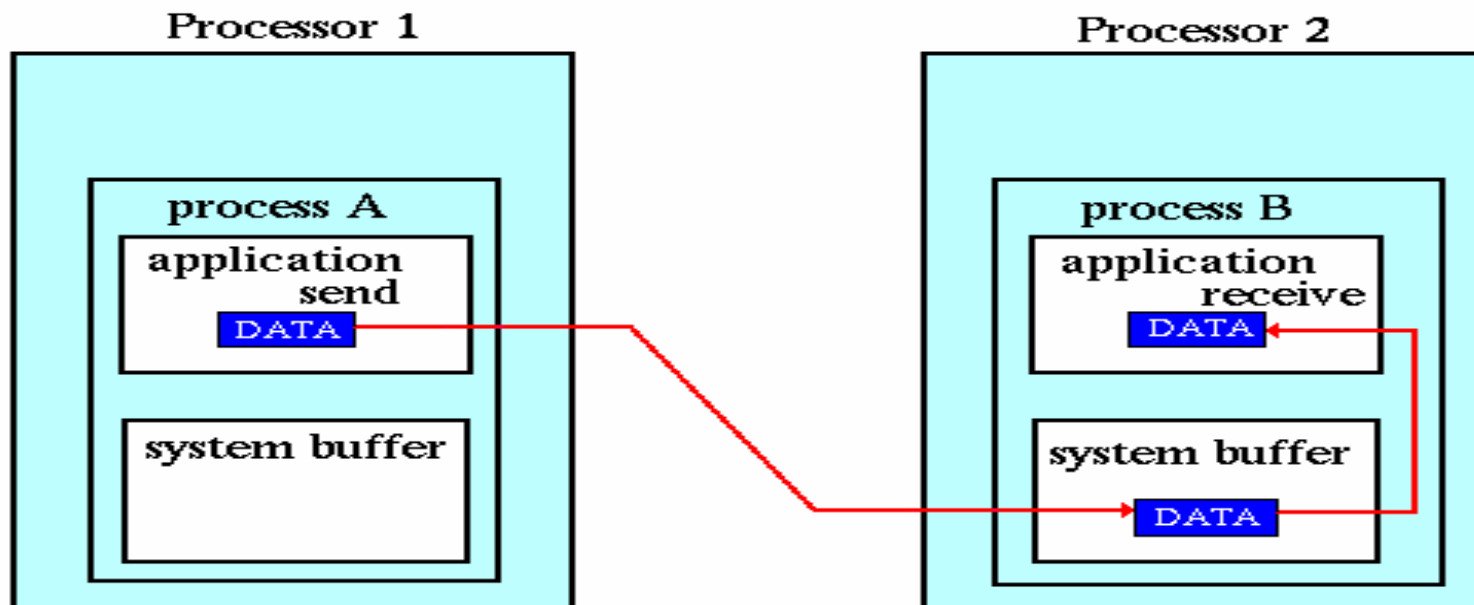
■ Необходимы уточнения процесса передачи



■ Требуется уточнить:

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

Схема выполнения операций передачи сообщений



Path of a message buffered at the receiving process

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 ea передающей стороне, так и на принимающей или на обеих сторонах
- Повышает производительность параллельной программы

Режимы выполнения операций передачи сообщений

- Режимы MPI-коммуникаций определяют, при каких условиях операции передачи завершаются
- Режимы могут быть блокирующими или неблокирующими
 - Блокирующие: возврат из функций передачи сообщений только по завершению коммуникаций
 - Неблокирующие (асинхронные): немедленный возврат из функций, пользователь должен контролировать завершение передач

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

MPI_FLOAT

MPI_DOUBLE

MPI_LONG_DOUBLE

Unsigned

MPI_UNSIGNED_CHAR

MPI_UNSIGNED_SHORT

MPI_UNSIGNED

MPI_UNSIGNED_LONG

Базовые MPI-типы данных (C)

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
- Односоторонние 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 comm, int*errorcode)*

завершает работу программы

Инициализация MPI

MPI_Init должна первым вызовом, вызывается только один раз

C:

```
int MPI_Init(int *argc, char ***argv)
```

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

Обработка ошибок MPI-функций

Определяется константой ***MPI_SUCCESS***

```
|  
int error;  
  
.....  
error = MPI_Init(&argc, &argv);  
If (error != MPI_SUCCESS)  
{  
fprintf (stderr, “ MPI_Init error \n”);  
return 1;  
  
}
```

MPI_Comm_size

Количество процессов в коммуникаторе

- **Размер коммуникатора**

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

Результат – число процессов

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

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-отправителя и получателя

```
int MPI_Comm_rank(MPI_Comm comm, int  
*rank)
```

Результат – номер процесса

Завершение MPI-процессов

- Никаких вызовов MPI функций после C:

int MPI_Finalize()

*int MPI_Abort (MPI_Comm_size(MPI_Comm comm, int*errorcode)*

Если какой-либо из процессов не выполняет MPI_Finalize, программа зависает.

Hello, MPI world! (2)

```
#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; }
```


Трансляция MPI-программ

- Трансляция

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

Например:

mpicc -o hw helloworld.c

- Запуск в интерактивном режиме

mpirun -np 128 hw

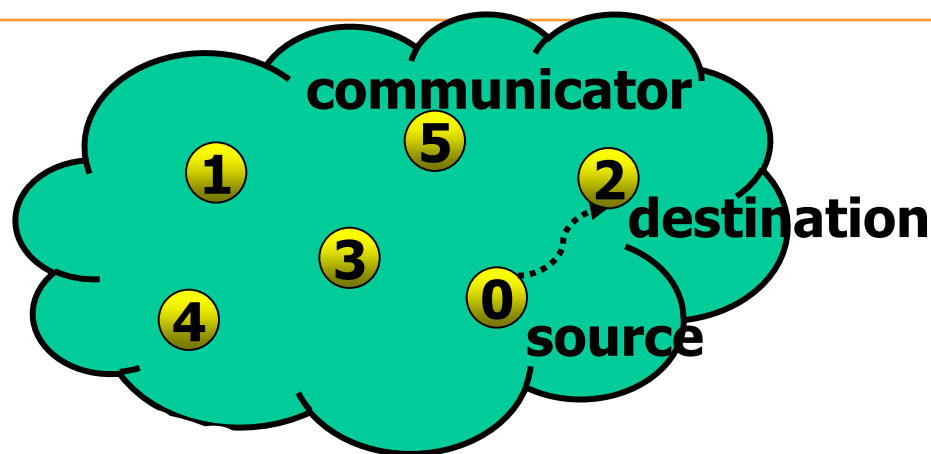
Взаимодействие «точка-точка»

- 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		
<u>MPI Bsend</u>	<u>MPI Bsend_init</u>	<u>MPI Buffer_attach</u>
<u>MPI Buffer_detach</u>	<u>MPI Cancel</u>	<u>MPI Get_count</u>
<u>MPI Get_elements</u>	<u>MPI Ibsend</u>	<u>MPI Iprobe</u>
<u>MPI Irecv</u>	<u>MPI Irsend</u>	<u>MPI Isend</u>
<u>MPI Issend</u>	<u>MPI Probe</u>	<u>MPI Recv</u>
<u>MPI Recv_init</u>	<u>MPI Request_free</u>	<u>MPI Rsend</u>
<u>MPI Rsend_init</u>	<u>MPI Send</u>	<u>MPI Send_init</u>
<u>MPI Sendrecv</u>	<u>MPI Sendrecv_replace</u>	<u>MPI Ssend</u>
<u>MPI Ssend_init</u>	<u>MPI Start</u>	<u>MPI Startall</u>
<u>MPI Test</u>	<u>MPI Test_cancelled</u>	<u>MPI Testall</u>
<u>MPI Testany</u>	<u>MPI Testsome</u>	<u>MPI Wait</u>
<u>MPI Waitall</u>	<u>MPI Waitany</u>	<u>MPI Waitsome</u>

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

- “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 коммутаторе (**comm**) .
- По завершению функции буфер может быть использован.

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

Address of
send buffer

Number of items
to send

Datatype of
each item

Rank of destination
process

Message tag

Communicator

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 коммутаторе **comm** или **MPI_ANY_SOURCE**.
- **status** содержит дополнительную информацию

MPI_Recv(buf, count, datatype, src, tag, comm, status)

Address of
receive buffer

Maximum number
of items to receive

Datatype of
each item

Rank of source
process

Message tag

Communicator

Status
after operation

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)
```


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)
```

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*

Информация о завершившемся приеме сообщения

- Возвращается функцией **`MPI_Recv`** через параметр **`status`**
- Содержит:
 - Source: *`status.MPI_SOURCE`*
 - Tag: *`status.MPI_TAG`*
 - Count: *`MPI_Get_count`*

Полученное сообщение

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

C:

```
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 );
```

Условия успешного взаимодействия «точка-точка»

- Отправитель должен указать правильный rank получателя
- Получатель должен указать верный rank отправителя
- Одинаковый коммутатор
- Тэги должны соответствовать друг другу
- Буфер у процесса-получателя должен быть достаточного объема

Совмещение отправки и приема сообщений

Для совмещения отправки и последующего приема сообщений MPI обеспечивает функции:

```
int MPI_Sendrecv(void *sendbuf, int sendcount,  
MPI_Datatype senddatatype, int dest, int sendtag, void *recvbuf,  
int recvcount, MPI_Datatype recvdatatype, int source, int recvtag,  
MPI_Comm comm, MPI_Status *status)
```

Если использовать один буфер:

```
int MPI_Sendrecv_replace(void *buf, int count,  
MPI_Datatype datatype, int dest, int sendtag,  
int source, int recvtag, MPI_Comm comm,  
MPI_Status *status)
```

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;
```


Неблокирующие коммуникации

Цель – уменьшение времени работы параллельной программы за счет совмещения вычислений и обменов.

Неблокирующие операции завершаются, не дожидаясь окончания передачи данных. В отличие от аналогичных блокирующих функций изменен критерий завершения операций – немедленное завершение.

Проверка состояния передач и ожидание завершения передач выполняются специальными функциями.

Неблокирующие функции

Non-Blocking Operation	MPI функции
Standard send	MPI_Isend
Synchronous send	MPI_Issend
Buffered send	MPI_Ibsend
Ready send	MPI_Irsend
Receive	MPI_Irecv


“I” : Immediate

Параметры неблокирующих операций

Datatype	Тип MPI_Datatype
Communicator	Аналогично блокирующим (тип MPI_Comm)
Request	Тип MPI_Request

- Параметр request задается при инициации неблокирующей операции
- Используется для проверки завершения операции

Совмещение блокирующих и неблокирующих операций

- Send и receive могут блокирующими и неблокирующими
- Блокирующий send может соответствовать неблокирующему receive, и наоборот, например,
MPI_Isend  ***MPI_Recv***
- Неблокирующий send может быть любого типа – synchronous, buffered, standard, ready

Задание 3.

- Разработать и реализовать параллельный алгоритм умножения плотной матрицы на вектор: $A * b$
- Распределение элементов матриц по процессам – ленточное, строчное.