

MPI

Message Passing Interface

Лекция 1

Что почитать

- Русский Internet ресурс
(<http://www.parallel.ru>)
- Лекции Воеводина
(<http://parallel.ru/parallel/vvv/mpi.html>)
- Стандарт MPI
(<http://www-unix.mcs.anl.gov/mpi/standard.html>)
- Jan Forster “Designing and Building Parallel programs” (<http://www-unix.mcs.anl.gov/dbpp/>)
- Лекция Вильяма Гроффа
(<http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html>)

Существующие реализации MPI

- MPICH (<http://www-unix.mcs.anl.gov/mpi/mpich>)
- LAM (<http://www.lam-mpi.org>)
- WMPI (<http://www.criticalsoftware.com/hpc>)
- OpenMPI (<http://www.open-mpi.org/>)
- Intel cluster tools
(<http://www.intel.com/cd/software/products/asmo-na/eng/cluster/>)
- IBM Parallel Operating Environment
(<http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp>)
- SUN HPC cluster tools
(<http://www.sun.com/products/hpc/communitysource/>)
- Scali (<http://www.scali.com/>)
- Microsoft Compute Cluster Server
(<http://windowshpc.net/default.aspx>)

Задачи MPI как среды программирования.

- Обеспечить интерфейс передачи сообщений в условиях разделённой памяти.
- Возможность писать приложения для гетерогенной многопроцессорной среды.
- Прозрачный механизм запуска параллельного приложения основанный на SPMD модели.

Задачи MPI как среды программирования.

- Понятный для пользователя механизм запуска параллельного приложения основанный на SPMD модели.
- Соккрытие от пользователя MPI всех деталей реализации конкретной физической среды межпроцессорных взаимодействий.

Компоненты MPI

- `mpirun` - Запуск программ на выполнение.

`mpirun -np 12 prog param1 param2 ...`

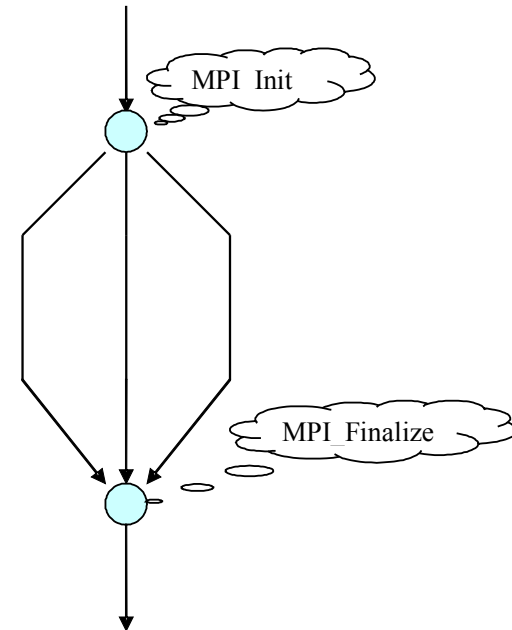
- `mpicc`, `mpif77`, `mpif90`, `mpiCC`, `mpicxx` – компиляторы.
- Средства отладки и построения трасс сообщений. (Например `jumpshot` в MPICH)

Основные понятия MPI.

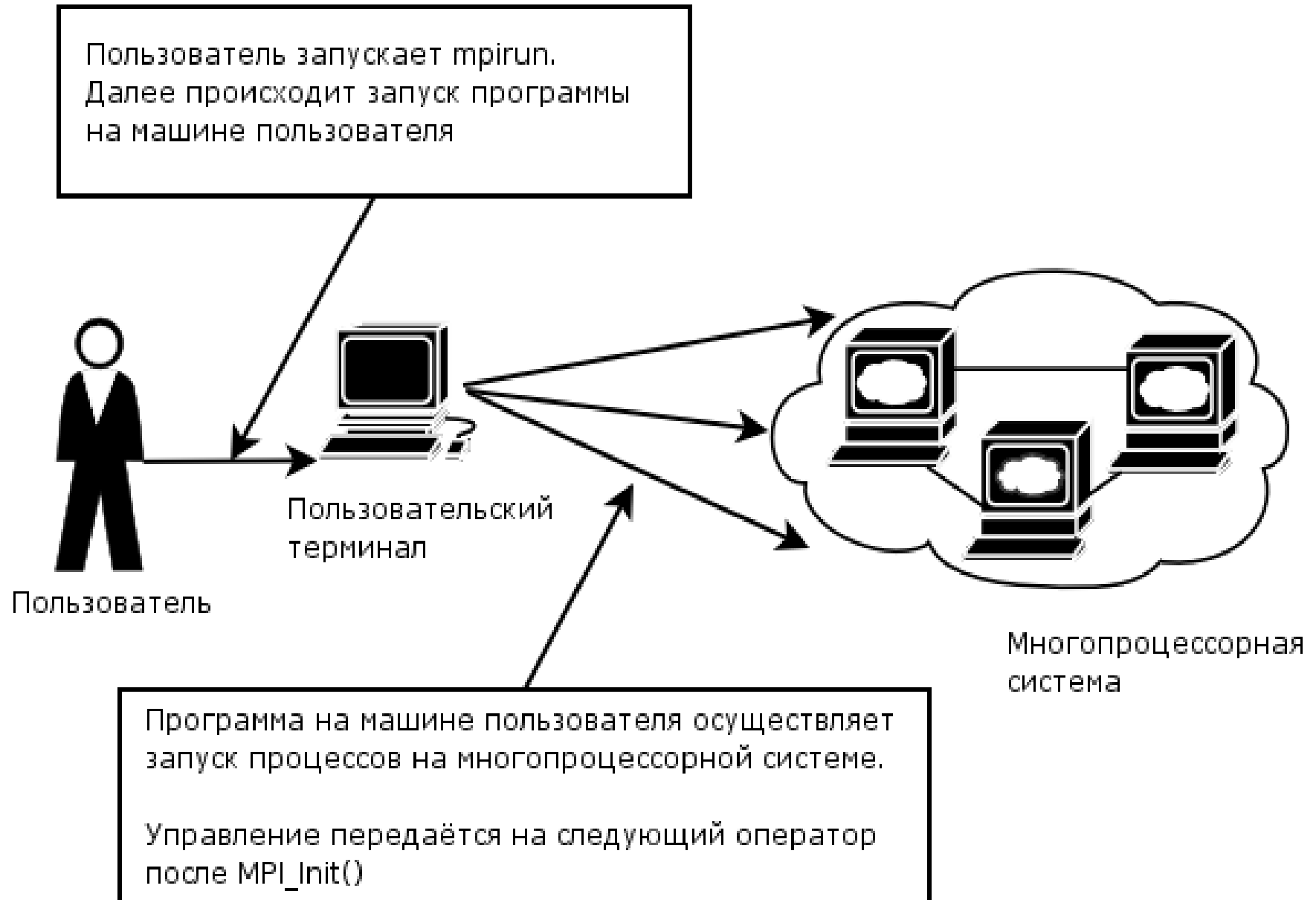
- Процесс (Нить)
- Коммуникатор и группы.
- Типы данных и манипуляции с данными (упаковка, распаковка и создание собственных типов.)
- Передача данных (блокирующая, не блокирующая, коллективные обмены данными.)
- Средства синхронизации.

Программа на MPI

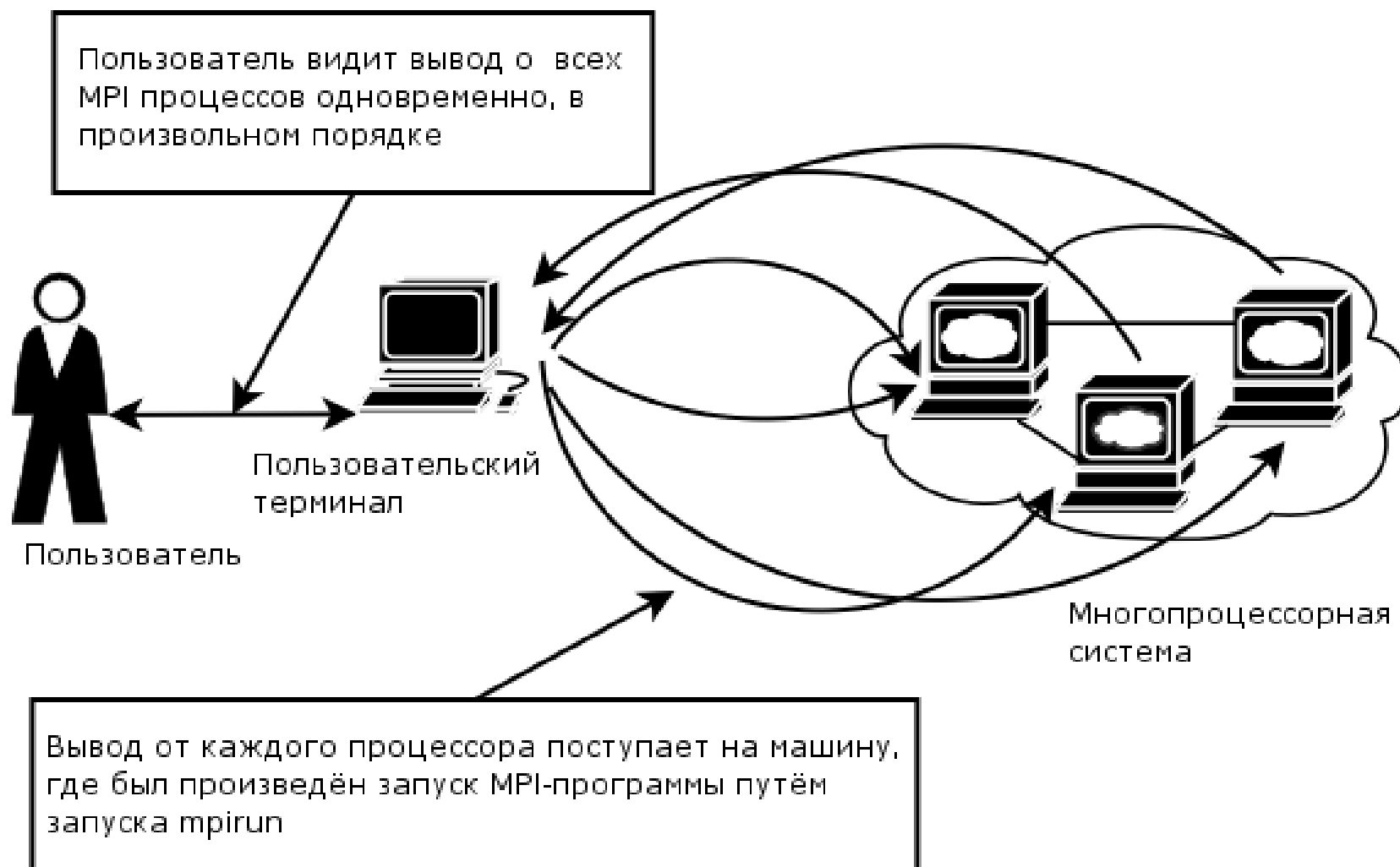
- Программа распадается на набор процессов (нитей).
- Ветвление производится по вызову:
`MPI_Init(int **argc, char ***argv);`
- Нить завершается по `MPI_Finalize();`
- Или завершается по `MPI_Abort();`



Процесс запуска MPI программы



Ввод-вывод в MPI



MPI_Finalize() и MPI_Abort()

- MPI_Finalize() – Завершает работу нити, освобождает процессор если он занят данной нитью, если вызван в управляющей нити, то блокирует нить и ждёт завершения остальных.
- MPI_Abort(comm,int status) – в случае вызова хотябы одной нитью, завершает все нити коммуникатора, где процесс операционной системы возвращает код завершения status.

Процессы

- Все нити изначально присоединены к коммуникатору `MPI_COMM_WORLD`
- Нить может выяснить число запущенных нитей при помощи функции `MPI_Comm_size(MPI_COMM_WORLD, &size);`
- Нить может выяснить свой порядковый номер `MPI_Comm_rank(MPI_COMM_WORLD, &rank);`
 - В MPI-1 число нитей строго фиксировано до момента запуска программы и в процессе не меняется. (В MPI-2 – это ограничение снято.)

Простейший пример программы на MPI

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

int main(int argc, char **argv)
{
    int size, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    printf("I am %d where processes %d. Hello\n", rank, size);

    MPI_Finalize();
    return 0;
}
```

Простейший пример программы на MPI

```
program main  
include 'mpif.h'  
begin
```

```
integer size, rank
```

```
call MPI_INIT(ierr)
```

```
call MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)
```

```
call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
```

```
print *, "I am ", rank, " of ", size, " processes. Hello World!!!"
```

```
call MPI_FINALIZE();
```

```
end
```

Обмен данными

- Передача типа точка-точка
(блокированная, неблокированная)
- Коллективные пересылки.
- Распределённые операции.

Блокированные обмены типа точка-точка.

- `MPI_Send(buf,count,datatype,dest,tag,comm)`
Данная процедура блокирует вызвавший её процесс, до тех пор, пока данные не будут приняты с другой стороны.

buf – собственно передаваемые данные

count – число ячеек передаваемых данных.

datatype – тип передаваемых данных (MPI_INT к примеру.)

dest – порядковый для указанного коммуникатора номер нити. Туда данные будут переданы.

tag – Числовая метка сообщения.

comm – коммуникатор, через который происходит передача данных.

Блокированные обмены типа точка-точка.

- `MPI_Recv(buf,count,datatype,dest,tag,comm,status)`
Блокирует вызвавший её процесс, пока данные от другого процесса не будут приняты целиком. Все параметры аналогичные `MPI_Send` но есть ещё `status`.

Статус структура содержит `tag` полученного сообщения, номер пославшей нити, размер сообщения. Используется совместно с `MPI_ANY_SOURCE` и `MPI_ANY_TAG`

Режимы отправки сообщений

- ***MPI_Ssend*** – немедленная небуферизованная отправка сообщения.
- ***MPI_Bsend*** – буферизованная отправка сообщения.
- ***MPI_Rsend*** – отправка сообщения при условии наличия информации о том, что принимающая сторона уже вызвала MPI_Recv к этому моменту.

Проблема синхронизации

- Тупики
- Гонки
- Неэффективность.

Синхронизация

- `MPI_Barrier(comm)` – останавливает процесс до тех пор, пока все процессы присоединённые к коммуникатору не вызовут барьер.

Пример: уравнение теплопроводности.

Рассмотрим задачу:

$$u_t = a^2 \cdot \Delta u(x) + f(x, t) \quad t \geq 0$$

$$u(x, t)|_{x \in G} = \varphi(t) \quad G = \overline{M} \setminus M \quad - \text{ граница области } - M.$$

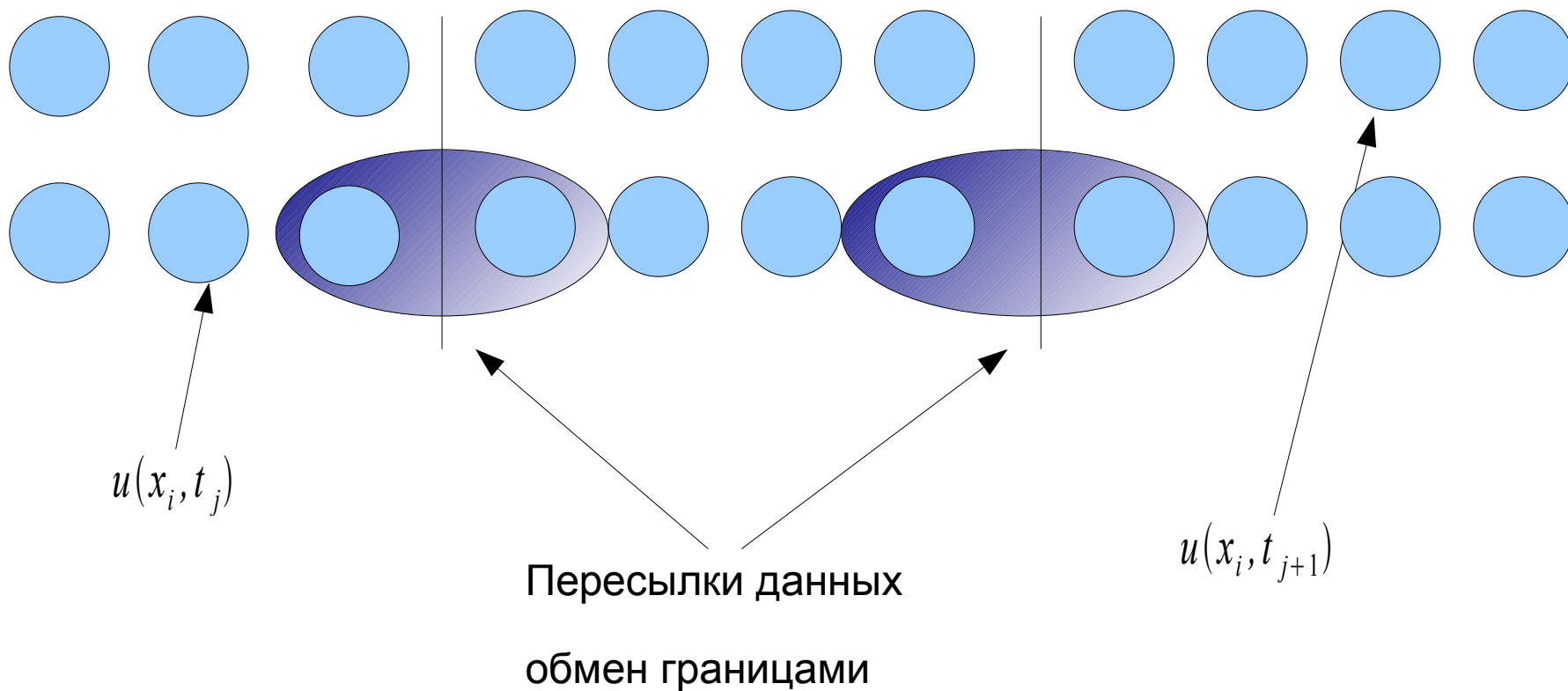
$$u(x, 0) = \psi(x) \quad x \in M$$

Пример: уравнение теплопроводности.

Сопоставим данной задаче разностную:

$$u(x_i, t_j + \Delta t) = \Delta t \cdot a^2 \cdot (u(x_{i-1}, t_j) - 2 * u(x_i, t_j) + u(x_{i+1}, t_j)) / \Delta x + f(x_i, t_j) \cdot \Delta t + u(x_i, t_j).$$

Способ распараллеливания.



Уравнение теплопроводности (неработающий вариант.)

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

- `#define T 100`
- `#define U 200`
- `#define a 4`

- `int main(int argc, char **argv)`
- `{`
- `int comm_rank, comm_size;`
- `double u[T][U]; /* 200 points by X and 100 by T and f=0 */`
- `int i, t;`
- `MPI_Status st;`

- `for(t=0; t<T; t++)`
- `{`
- `u[t][0]=1;`
- `u[t][U-1]=100;`
- `}`

- `for(i=1; i<U-1; i++)`
- `{`
- `u[0][i]=100/(double)(2*i);`
- `}`

- `MPI_Init(&argc, &argv);`
- `MPI_Comm_size(MPI_COMM_WORLD, &comm_size);`
- `MPI_Comm_rank(MPI_COMM_WORLD, &comm_rank);`

Уравнение теплопроводности (неработающий вариант.)

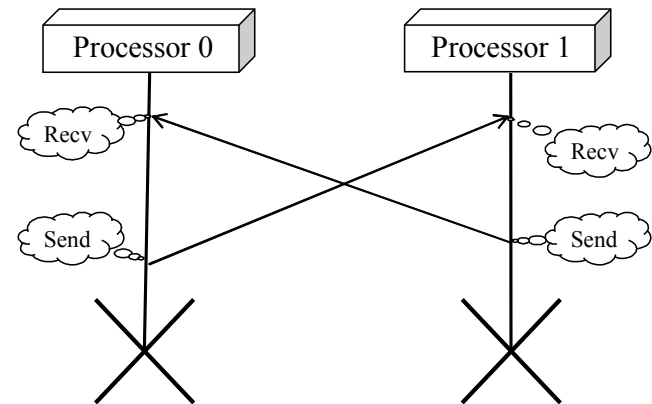
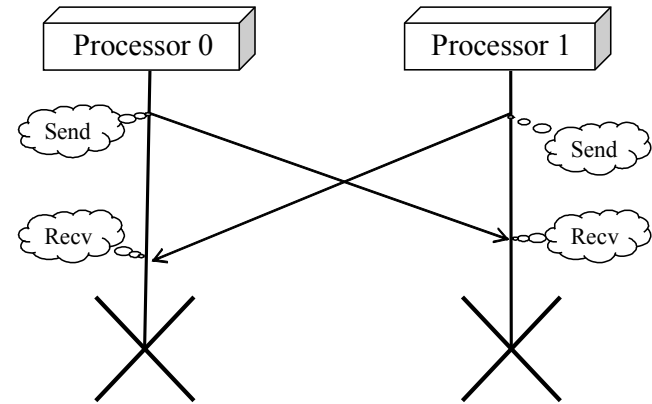
```
•   for(t=0;t<T-1;t++)
•   {
•   if((comm_rank!=comm_size-1)&&(comm_rank!=0))
•   {
•   MPI_Ssend(&u[t][(U/comm_size)*(comm_rank)],1,MPI_DOUBLE,comm_rank-1,0,MPI_COMM_WORLD);
•   MPI_Ssend(&u[t][(U/comm_size)*(comm_rank+1)],1,MPI_DOUBLE,comm_rank+1,1,MPI_COMM_WORLD);
•   MPI_Recv(&u[t][(U/comm_size)*(comm_rank)-1],1,MPI_DOUBLE,comm_rank-1,1,MPI_COMM_WORLD,&st);
•   MPI_Recv(&u[t][(U/comm_size)*(comm_rank+1)+1],1,MPI_DOUBLE,comm_rank+1,0,MPI_COMM_WORLD,&st);
•   }
•   if(comm_rank==0)
•   {
•   MPI_Ssend(&u[t][(U/comm_size)*(comm_rank)],1,MPI_DOUBLE,1,1,MPI_COMM_WORLD);
•   MPI_Recv(&u[t][(U/comm_size)*(comm_rank+1)+1],1,MPI_DOUBLE,1,0,MPI_COMM_WORLD,&st);
•   }
•   if(comm_rank==comm_size-1)
•   {
•   MPI_Ssend(&u[t][(U/comm_size)*(comm_rank)],1,MPI_DOUBLE,comm_rank-1,0,MPI_COMM_WORLD);
•   MPI_Recv(&u[t][(U/comm_size)*(comm_rank)-1],1,MPI_DOUBLE,comm_rank-1,1,MPI_COMM_WORLD,&st);
•   }
•   }
```

Уравнение теплопроводности (неработающий вариант.)

```
• if(comm_rank!=comm_size-1)
• {
•   for(i=(U/comm_size)*(comm_rank);i<(U/comm_size)*(comm_rank+1);i++)
•   {
•     u[t+1][i]=a*a*(u[t][i-1]-2*u[t][i]+u[t][i+1])+u[t][i];
•     printf("u[%d][%d]=%lf\n",t+1,i,u[t+1][i]);
•   }
• }
• else
• {
•   for(i=(U/comm_size)*(comm_rank-1);i<U;i++)
•   {
•     u[t+1][i]=a*a*(u[t][i-1]-2*u[t][i]+u[t][i+1])+u[t][i];
•     printf("u[%d][%d]=%lf\n",t+1,i,u[t+1][i]);
•   }
• }
• } /* END FOR through T */
•
• MPI_Finalize();
• return 0;
• }
```

Тупики (deadlocks)

- В первом случае никогда не вызовется `MPI_Recv`
- Во втором случае никогда не вызовется `MPI_Send`



Уравнение теплопроводности (плохо работающий вариант.)

- for(t=0;t<T-1;t++)
- {
- if(comm_rank==0)
- {
- MPI_Send(&u[t][(U/comm_size)*(comm_rank)],1,MPI_DOUBLE,1,1,MPI_COMM_WORLD);
- MPI_Recv(&u[t][(U/comm_size)*(comm_rank+1)+1],1,MPI_DOUBLE,1,0,MPI_COMM_WORLD,&st);
- }
- if((comm_rank!=comm_size-1)&&(comm_rank!=0))
- {
- MPI_Recv(&u[t][(U/comm_size)*(comm_rank)-1],1,MPI_DOUBLE,comm_rank-1,1,MPI_COMM_WORLD,&st);
- MPI_Send(&u[t][(U/comm_size)*(comm_rank+1)],1,MPI_DOUBLE,comm_rank+1,1,MPI_COMM_WORLD);
- MPI_Recv(&u[t][(U/comm_size)*(comm_rank+1)+1],1,MPI_DOUBLE,comm_rank+1,0,MPI_COMM_WORLD,&st);
- MPI_Send(&u[t][(U/comm_size)*(comm_rank)],1,MPI_DOUBLE,comm_rank-1,0,MPI_COMM_WORLD);
- }
- }
- if(comm_rank==comm_size-1)
- {
- MPI_Recv(&u[t][(U/comm_size)*(comm_rank)-1],1,MPI_DOUBLE,comm_rank-1,1,MPI_COMM_WORLD,&st);
- MPI_Send(&u[t][(U/comm_size)*(comm_rank)],1,MPI_DOUBLE,comm_rank-1,0,MPI_COMM_WORLD);
- }

Пример с гонками.

- `if(comm_rank!=0)`
- `for(i=0;i<100;i++)`
- `{`
- `MPI_Send(&i,1,MPI_INT,0,0,MPI_COMM_WORLD);`
- `}`
- `if(comm_rank==0)`
- `{`
- `for(i=0;i<100*(comm_size-1);i++)`
- `{`
- `MPI_Recv(&res,1,MPI_INT,MPI_ANY_SOURCE,0,`
- `MPI_COMM_WORLD,&st);`
- `printf("(%d,%d)\n",res,st.MPI_SOURCE);`
- `}`
- `}`

Приём данных неизвестного размера.

- `MPI_Probe(source,tag,comm,status)` – Блокирует процесс до тех пор, пока не будет возможно получить соответствующие данные с помощью `MPI_Recv`. Данная процедура заполняет `status`.
- `MPI_Get_count(status,datatype,count)` – в `count` выдаёт число элементов заданного типа данных в сообщении.

Пример приёма сообщений неизвестного размера.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv)
{
    int size, rank;

    int count;
    MPI_Status status;
    char *buf;

    MPI_Init(&argc, &argv); /* Инициализируем библиотеку */
    MPI_Comm_size(MPI_COMM_WORLD, &size); /* Узнаем количество задач в запущенном приложении... */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* ...и свой собственный номер: от 0 до (size-1) */

    if ((size > 1) && (rank == 0))
    {
        /* задача с номером 0 отправляет сообщение */
        MPI_Send(argv[0], strlen(argv[0])+1, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
        printf("Sent to process 1: \"%s\"\n", argv[0]);
    }
    else if ((size > 1) && (rank == 1))
    {
        /* задача с номером 1 получает сообщение */
        MPI_Probe(0, 1, MPI_COMM_WORLD, &status);
        MPI_Get_count(&status, MPI_CHAR, &count);
        buf = (char *) malloc(count * sizeof(char));
        MPI_Recv(buf, count, MPI_CHAR, 0, 1, MPI_COMM_WORLD, &status);
        printf("Received from process 0: \"%s\"\n", buf);
    }

    MPI_Finalize(); /* Все задачи завершают выполнение */
    return 0;
}
```

Неблокированные обмены.

- На фоне приёма и передачи данных могут производиться вычисления.
- `MPI_Isend(buf, count, datatype, dest, msgtag, comm, request)`
- `MPI_Irecv(buf, count, datatype, dest, msgtag, comm, request)` – нет статуса!
- В отличие от блокированного `send` появляется выходной параметр `request` – идентификатор послыки, по нему затем можно узнать состояние обмена данными. (Аналогично для `MPI_Irecv`)

Неблокированные обмены.

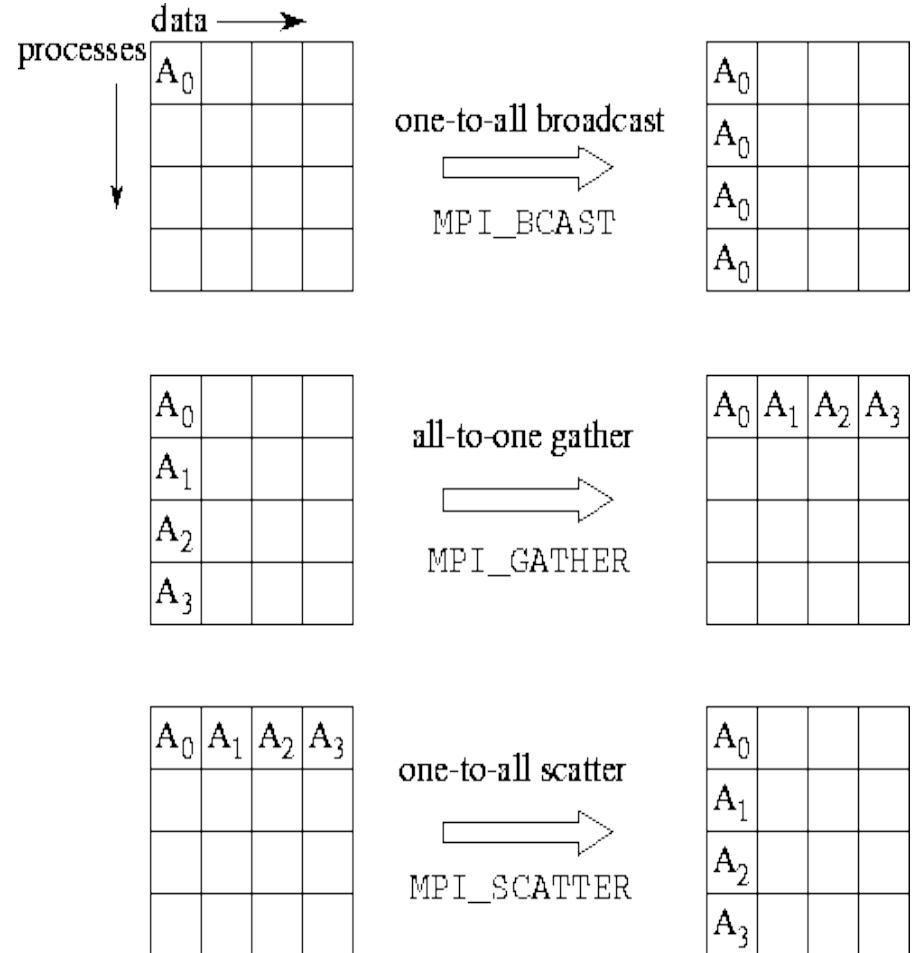
- `MPI_Wait(request,status)` Ждёт завершения одного обмена и инициализирует по окончании статус.
- `MPI_Waitall(count,requests,statuses)` – аналогична `MPI_Wait` только сразу для `count` обменов.
- `MPI_Waitany(count,requests,index,statuses)` – в `index` соберётся порядковый номер среди `requests` завершившегося обмена.

Неблокированные обмены.

- `MPI_Test(request, flag, status)` – В отличие от `MPI_Wait` не блокирует процесс, а возвращает 0 в случае, если обмен не завершился и 1 если завершился.
- `MPI_Testany` – аналогично `MPI_Waitany`
- `MPI_Testall` – аналогично `MPI_Waitall`
- `MPI_Iprobe` – Не блокирует процесс. Аналогично `MPI_Probe` заполняет `status`, но как и в случае с `MPI_Test` в `flag` возвращается 1 если данные готовы к приёму с помощью `MPI_Recv`.

Коллективные обмены

- `MPI_Bcast(buf,count,datatype,source,comm)` – распространяет буфер по всем процессам присоединённым к коммутатору.
- `MPI_Gather(sbuf,scount,stype,rbuf,rcount, rtype, dest,comm)` - собирает значения `sbuf` с процессов коммутатора в `rbuf` на процессе с номером `dest`.
- `MPI_Scatter` – действие противоположное к `MPI_Gather` содержимое `sbuf` с `source` процесса по другим процессам.
- `MPI_Alltoall` – действие аналогично транспонированию матрицы.



Распределённые операции.

- `MPI_Reduce(sbuf,rbuf,count,datatype,op,root,comm)` – Выполняет операцию `op` над соответствующими элементами каждого процесса, и результат запишет в процесс с номером `root`.
- `MPI_Allreduce(sbuf,rbuf,count,datatype,op,comm)` – делает тоже самое, но результат заносит во все процессы сразу.

