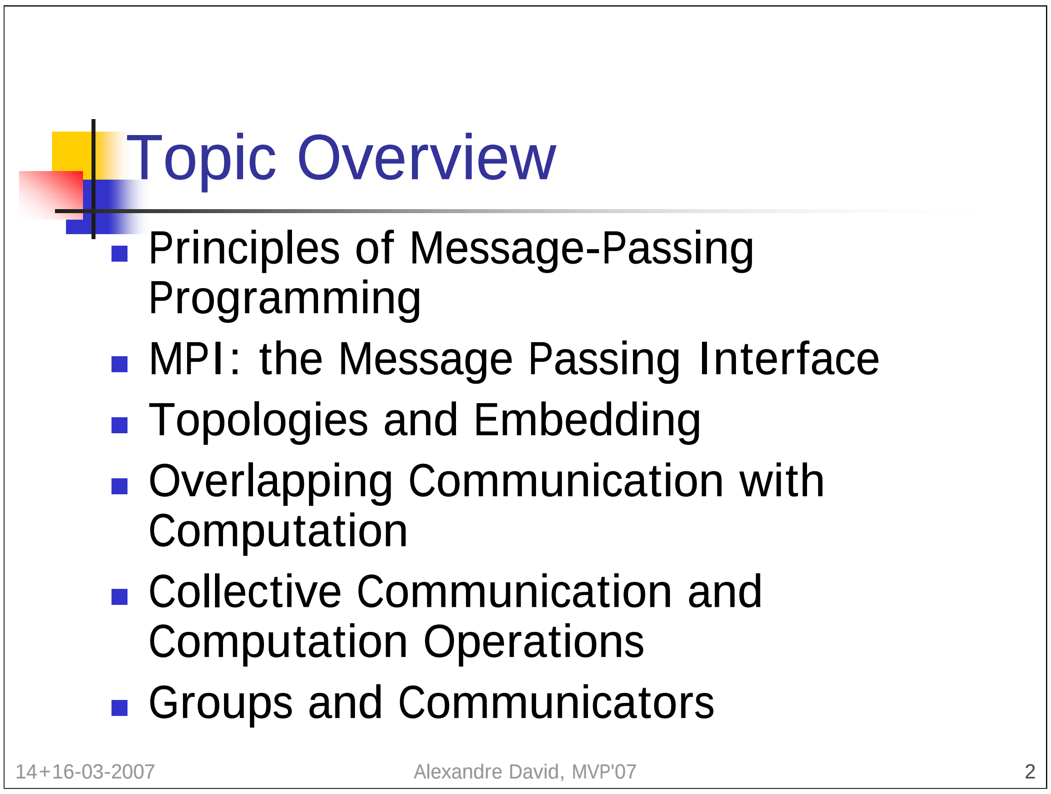
Programming Using the Message-Passing Paradigm (Chapter 6)

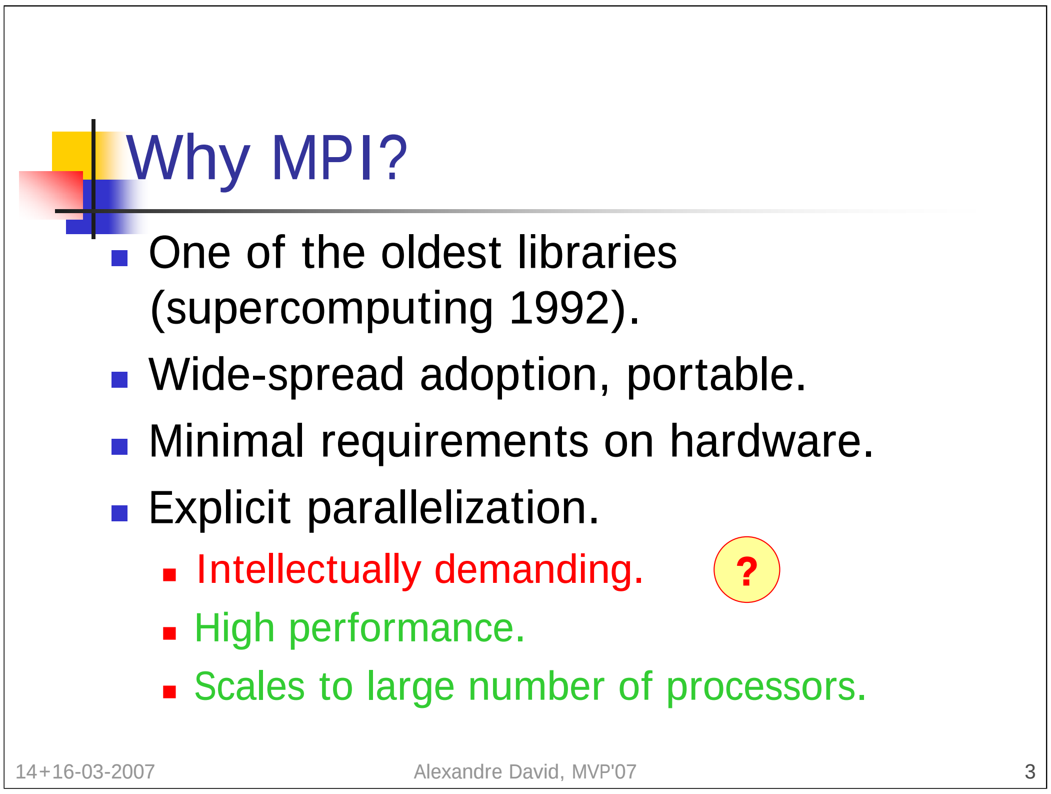
Alexandre David
B2-206



Topic Overview

- Principles of Message-Passing Programming
- MPI: the Message Passing Interface
- Topologies and Embedding
- Overlapping Communication with Computation
- Collective Communication and Computation Operations
- Groups and Communicators

Put in practice some theory we have seen so far.

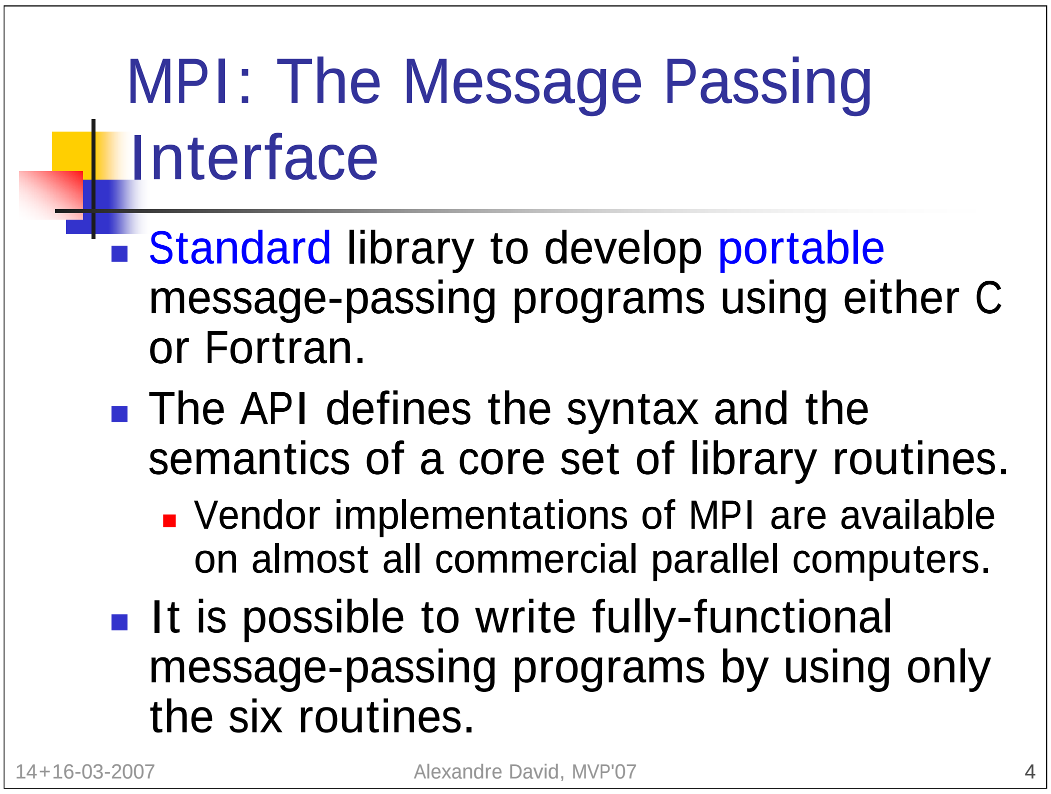


Why MPI?

- One of the oldest libraries (supercomputing 1992).
- Wide-spread adoption, portable.
- Minimal requirements on hardware.
- Explicit parallelization.
  - Intellectually demanding. ?
  - High performance.
  - Scales to large number of processors.

14+16-03-2007 Alexandre David, MVP'07 3

Remember previous lectures: The minimal requirement is a bunch of computers connected on a network.

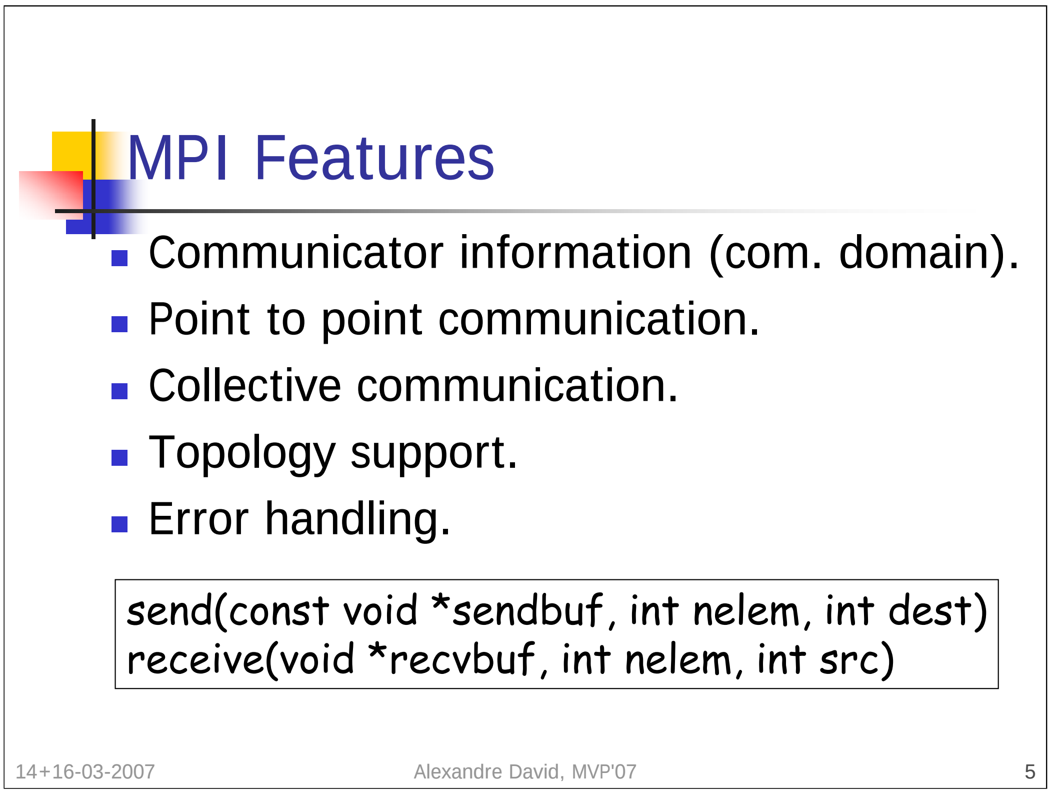


MPI: The Message Passing Interface

- Standard library to develop portable message-passing programs using either C or Fortran.
- The API defines the syntax and the semantics of a core set of library routines.
  - Vendor implementations of MPI are available on almost all commercial parallel computers.
- It is possible to write fully-functional message-passing programs by using only the six routines.

14+16-03-2007 Alexandre David, MVP'07 4

In the early time of parallel computing every vendor had its incompatible message-passing library with syntactic and semantic differences. Programs were not portable (or required significant efforts to port them). MPI was designed to solve this problem.

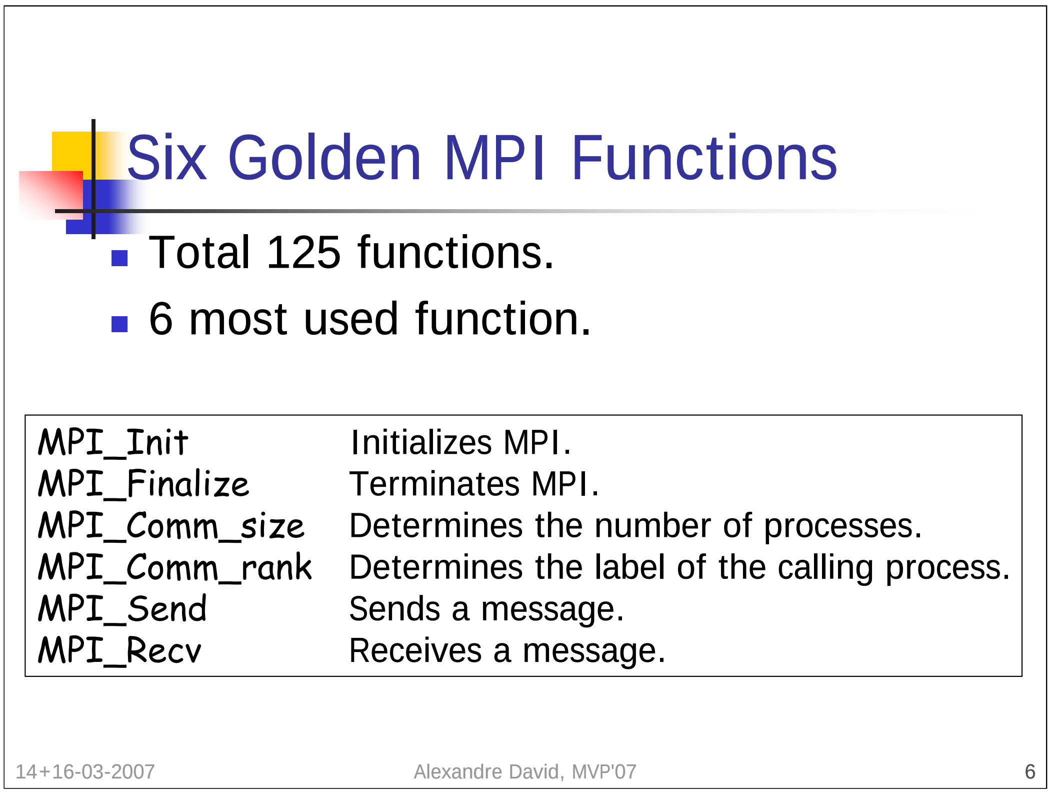


MPI Features

- Communicator information (com. domain).
- Point to point communication.
- Collective communication.
- Topology support.
- Error handling.

```
send(const void *sendbuf, int nelem, int dest)
receive(void *recvbuf, int nelem, int src)
```

And you can map easily these practical concepts to theory we have been studying. In summary send/receive are the most important primitives.



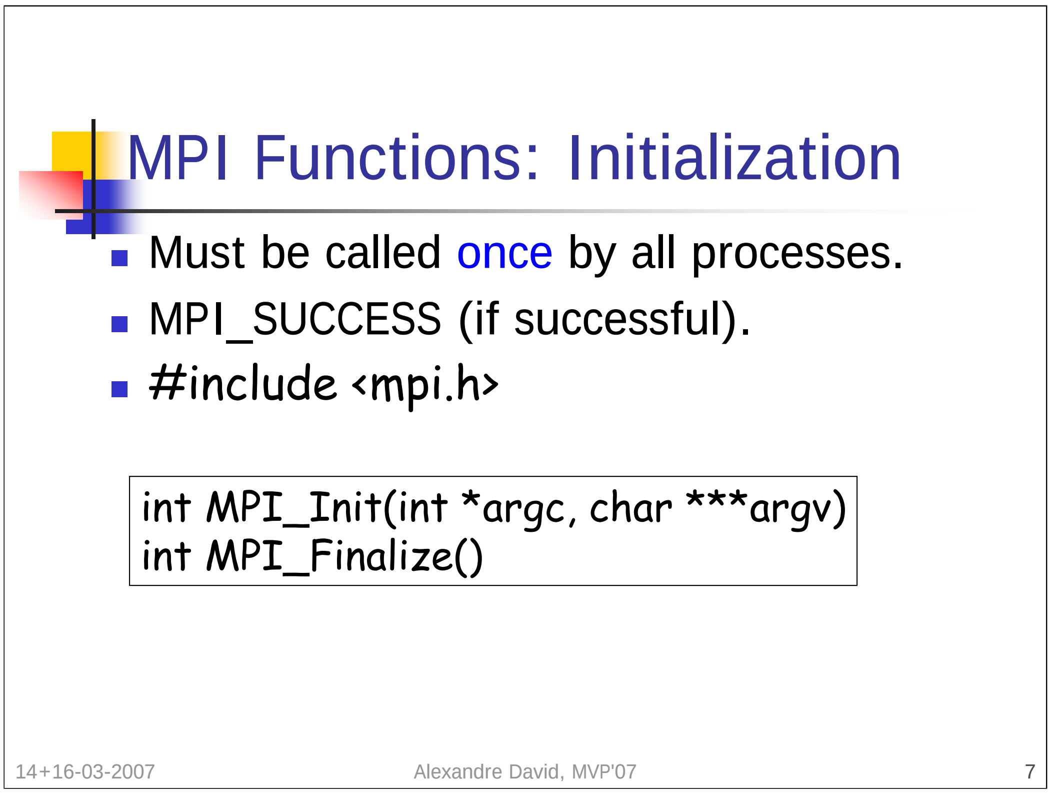
Six Golden MPI Functions

- Total 125 functions.
- 6 most used function.

MPI_Init	Initializes MPI.
MPI_Finalize	Terminates MPI.
MPI_Comm_size	Determines the number of processes.
MPI_Comm_rank	Determines the label of the calling process.
MPI_Send	Sends a message.
MPI_Recv	Receives a message.

14+16-03-2007 Alexandre David, MVP'07 6

Let’s have some taste of MPI.



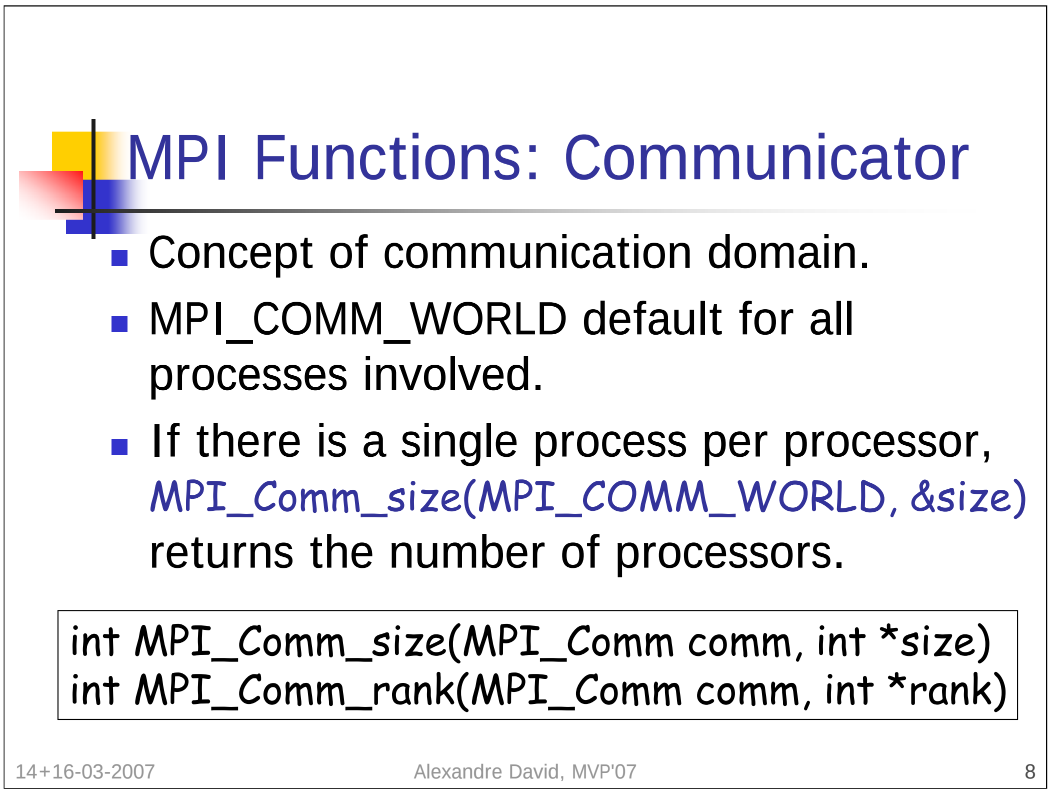
MPI Functions: Initialization

- Must be called once by all processes.
- MPI_SUCCESS (if successful).
- #include <mpi.h>

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

MPI_Init initializes the MPI environment. MPI_Finalize performs clean-up tasks, no MPI calls after that (not even MPI_Init). As for MPI_Init, MPI_Finalize must be called by all processes.

Arguments of MPI_Init: command line arguments. Arguments will be processed and removed because the program is run within an environment (mpirun) that is sending a bunch of special arguments to your program. **Exercise**: You can try to print all the arguments of your program before calling MPI_Init to see them if you want.



MPI Functions: Communicator

- Concept of communication domain.
- MPI_COMM_WORLD default for all processes involved.
- If there is a single process per processor, `MPI_Comm_size(MPI_COMM_WORLD, &size)` returns the number of processors.

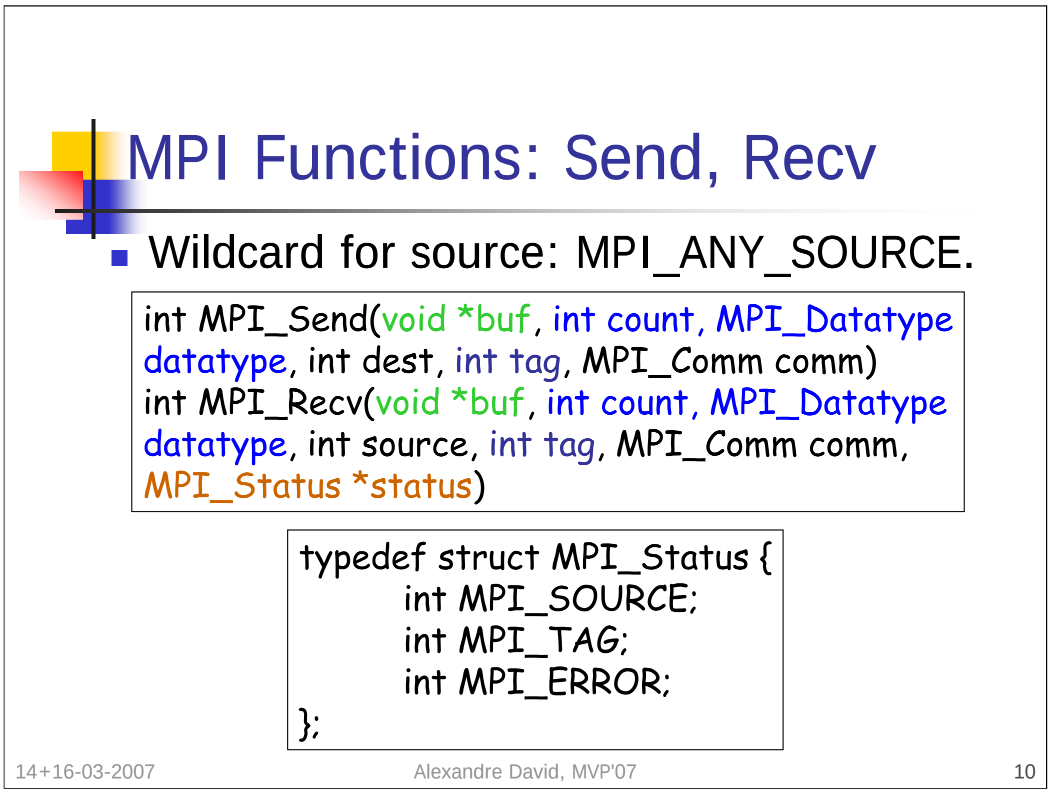
```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Communication domain = set of processes allowed to communicate with each other. Processes may belong to different communicators.

The rank is an int[0..comm_size-1]. Processes calling these functions must belong to the appropriate communicator otherwise error!

Hello World!

```
#include <mpi.h>
int main(int argc, char *argv[])
{
        int npes, myrank;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &npes);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        printf("From process %d out of %d, Hello world!\n",
                myrank, npes);
        MPI_Finalize();
        return 0;
}
```



MPI Functions: Send, Recv

- Wildcard for source: MPI_ANY_SOURCE.

```
int MPI_Send(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype
datatype, int source, int tag, MPI_Comm comm,
MPI_Status *status)
```

```
typedef struct MPI_Status {
        int MPI_SOURCE;
        int MPI_TAG;
        int MPI_ERROR;
};
```

14+16-03-2007 Alexandre David, MVP'07 10

The tag is used to distinguish different types of messages. Wildcard for tag: MPI_ANY_TAG.

Receiver side: size of the buffer specified. Received message size ≤ size of this buffer. If the received message is larger the function returns MPI_ERR_TRUNCATE.

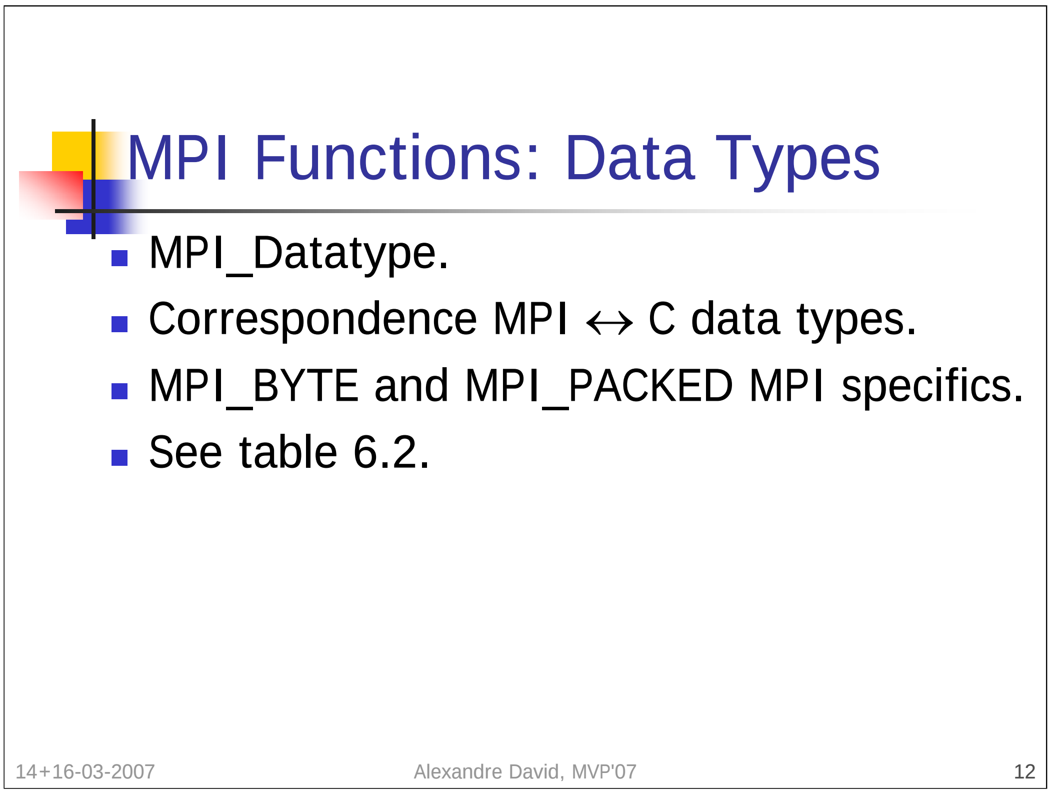
Status: MPI_SOURCE and MPI_TAG most useful when wildcards are used.

MPI_Recv is a **blocking** (buffered) receive operation. Different versions of send are available but the buffer argument must always be safe for overriding upon completion of the call. Programs should be **safe** in the sense that they should not depend on a particular implementation of send.

Length of Received Message

- Not directly accessible.
- Reminder: The returned int says if the call was successful or not.

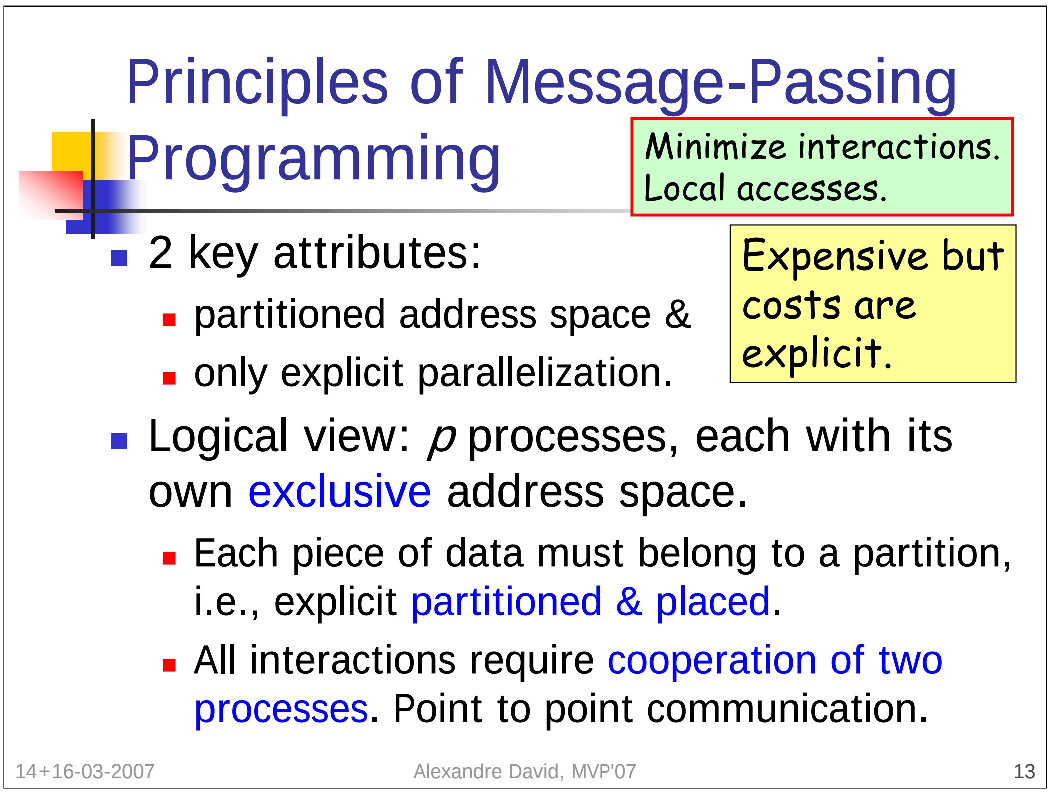
```
int MPI_Get_count(MPI_Status *status,
MPI_Datatype datatype, int *count)
```



MPI Functions: Data Types

- MPI_Datatype.
- Correspondence MPI ↔ C data types.
- MPI_BYTE and MPI_PACKED MPI specifics.
- See table 6.2.

MPI_PACKED provides marshalling.



Principles of Message-Passing Programming

Minimize interactions.
Local accesses.

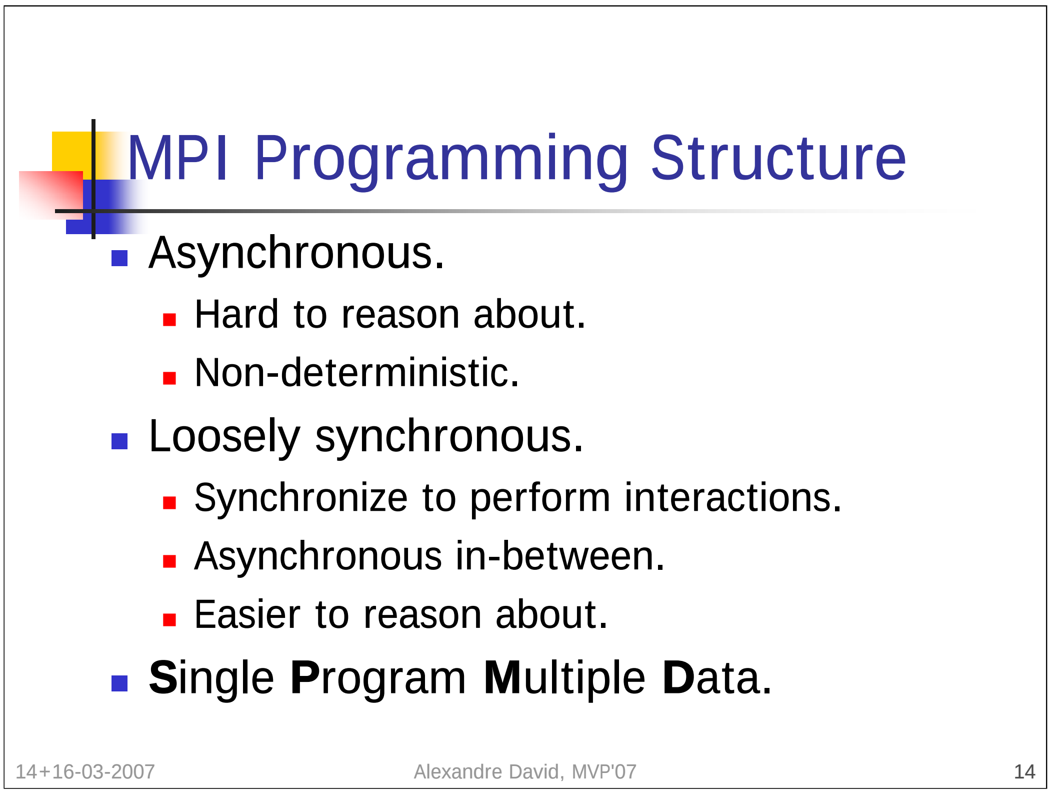
Expensive but costs are explicit.

- 2 key attributes:
  - partitioned address space &
  - only explicit parallelization.
- Logical view: *p* processes, each with its own exclusive address space.
  - Each piece of data must belong to a partition, i.e., explicit partitioned & placed.
  - All interactions require cooperation of two processes. Point to point communication.

14+16-03-2007 Alexandre David, MVP'07 13

Partitioning: add complexity but **encourages locality of access** (critical for performance as mentioned early in the course). Requirement on cooperation for read-only or read/write access adds complexity too but the programmer is fully aware of all the costs and will think about algorithms that **minimize interaction costs**.

Explicit parallelization: the programmer has to think more but result is more scalable.

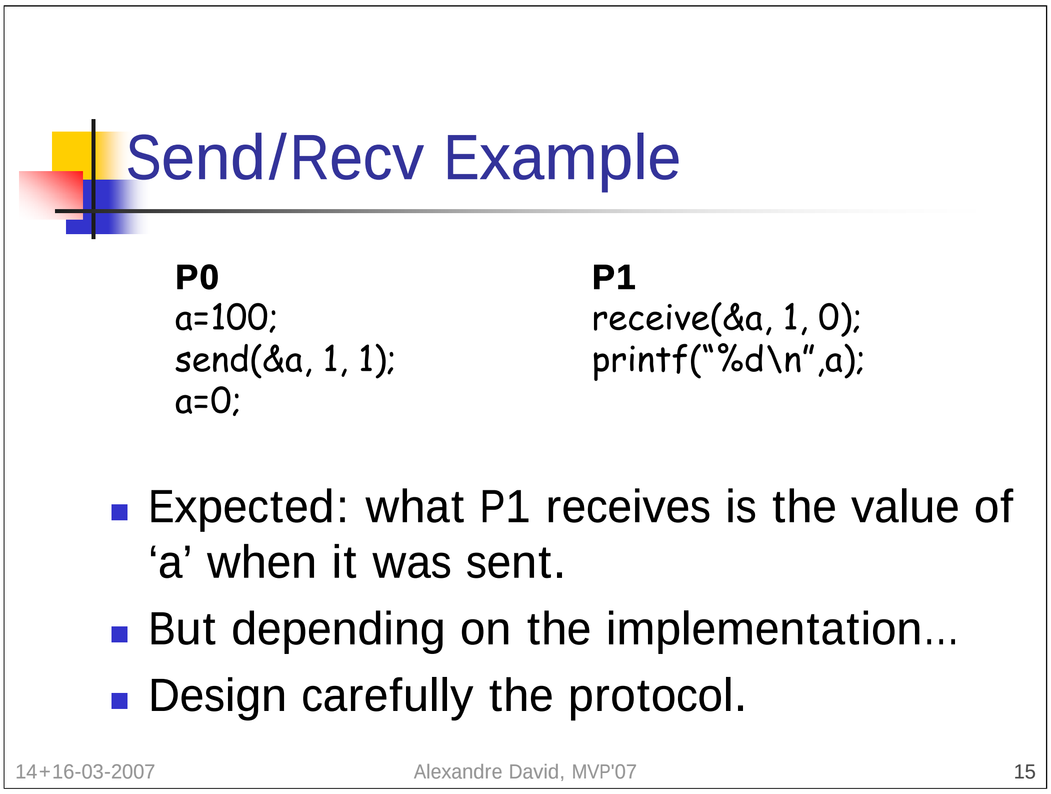


MPI Programming Structure

- Asynchronous.
  - Hard to reason about.
  - Non-deterministic.
- Loosely synchronous.
  - Synchronize to perform interactions.
  - Asynchronous in-between.
  - Easier to reason about.
- **S**ingle **P**rogram **M**ultiple **D**ata.

Asynchronous paradigm: All concurrent tasks execute asynchronously. Good compromise is to synchronize sometimes.

Most MPI programs are written following the SPMD programming model, i.e., the same code is executed on all the processors (with some exceptions like the “root” process).

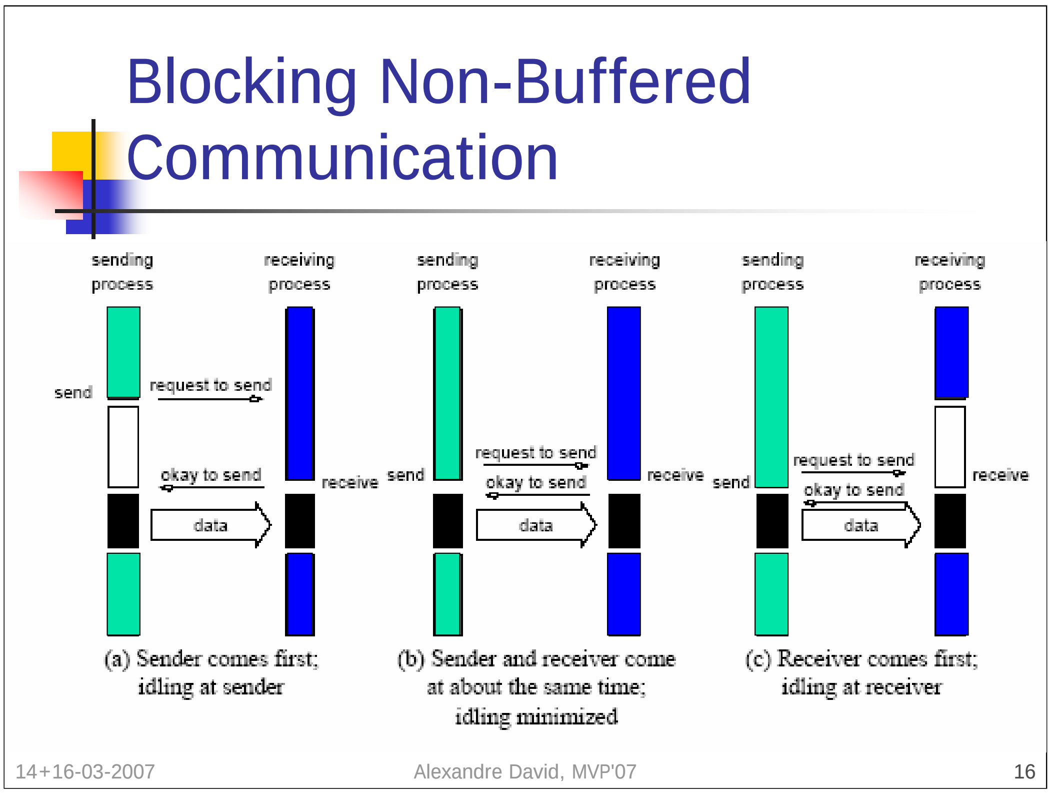


Send/Recv Example

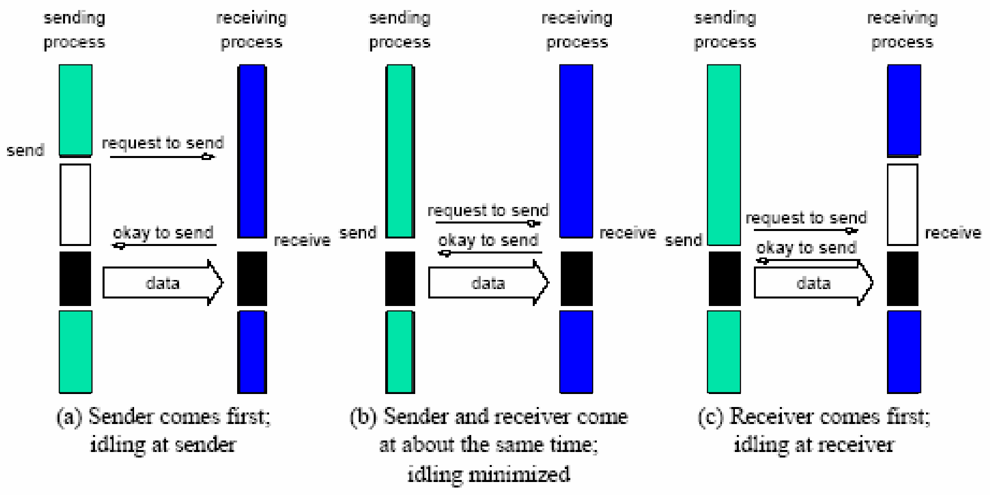
```
P0                      P1
a=100;                  receive(&a, 1, 0);
send(&a, 1, 1);         printf("%d\n",a);
a=0;
```

- Expected: what P1 receives is the value of ‘a’ when it was sent.
- But depending on the implementation...
- Design carefully the protocol.

What you see may not be what you get!



Blocking Non-Buffered Communication



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

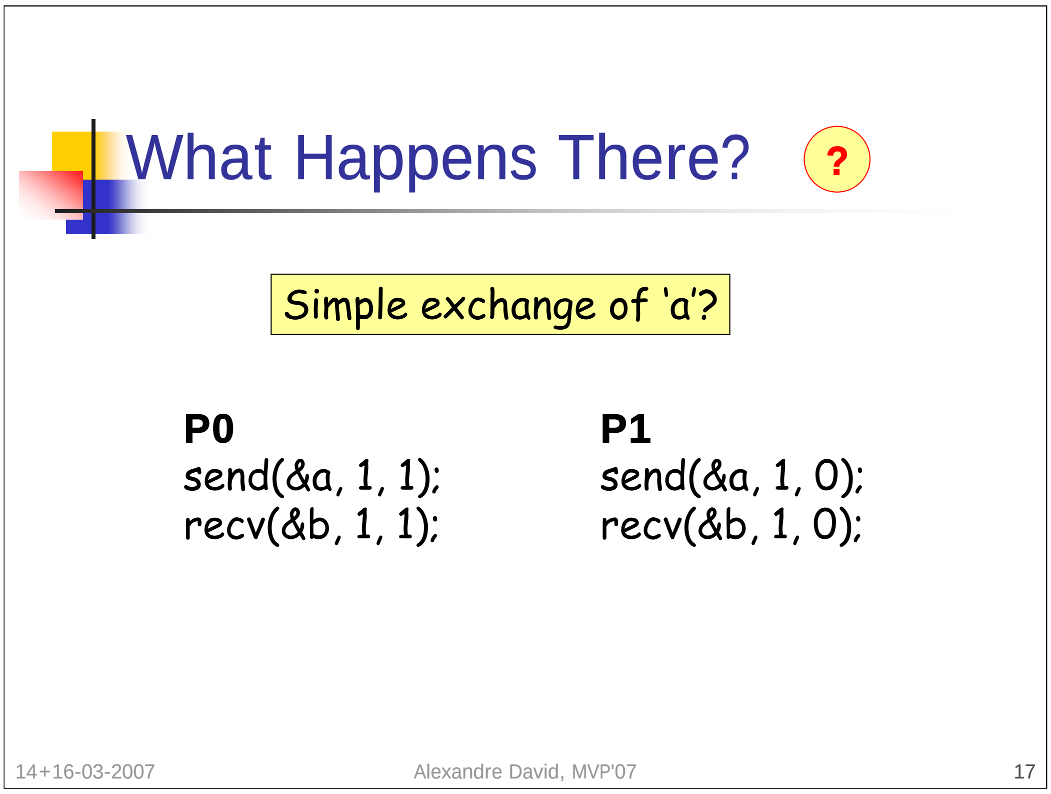
(c) Receiver comes first; idling at receiver

14+16-03-2007 Alexandre David, MVP'07 16

Simple method is to block & return only when it is safe.

Major issues: **idling** and **deadlock**.

How to improve: **Buffered blocking** = copy data and returns when the copy is completed. Reduce idling but copy overhead. Buffer the data at the receiver's side too.



What Happens There? ?

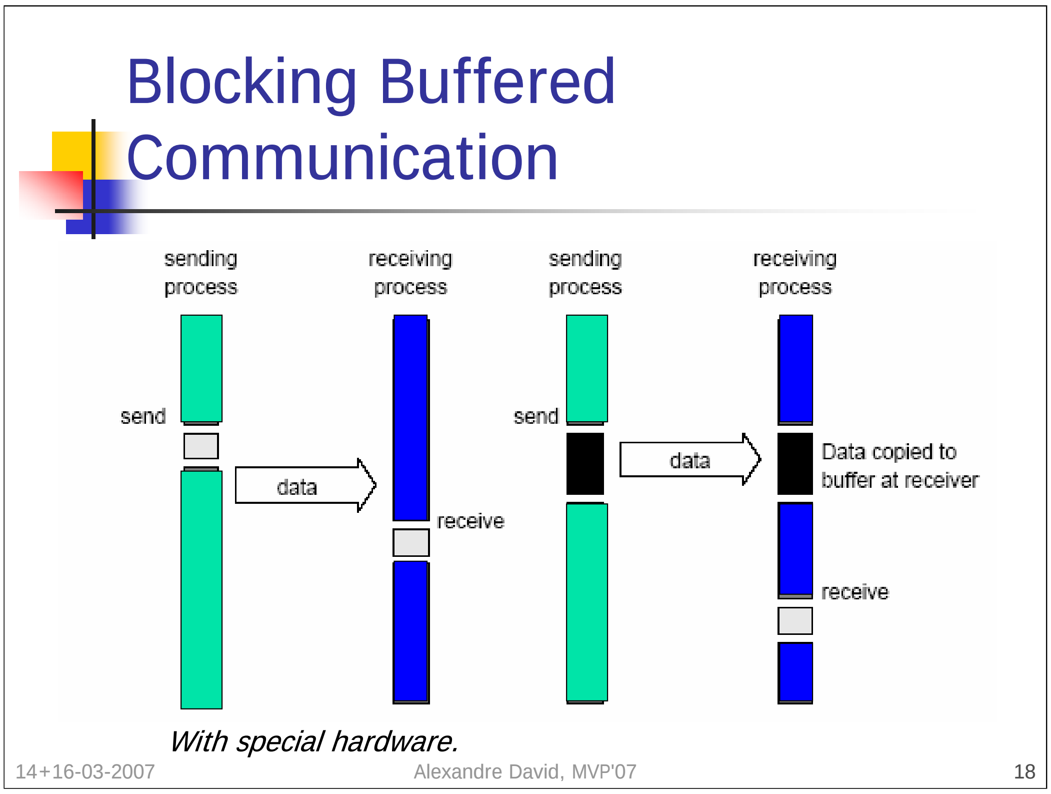
Simple exchange of 'a'?

P0
send(&a, 1, 1);
recv(&b, 1, 1);

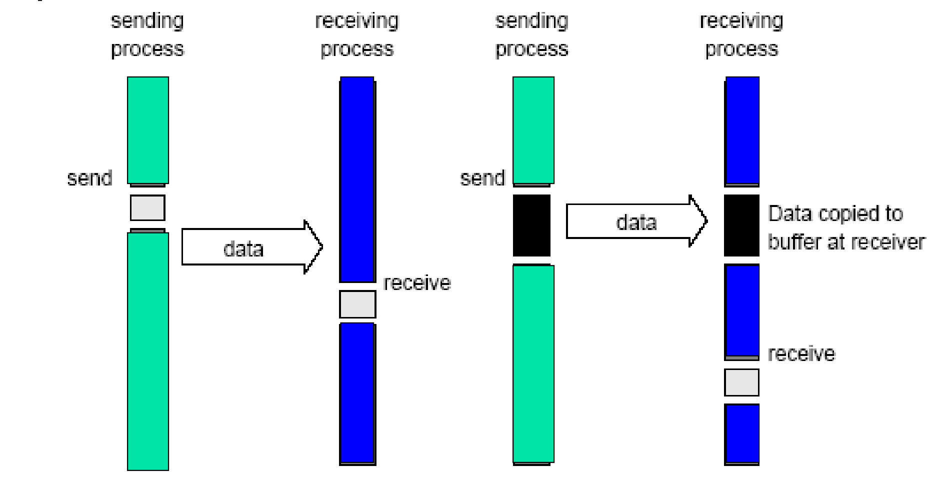
P1
send(&a, 1, 0);
recv(&b, 1, 0);

14+16-03-2007 Alexandre David, MVP'07 17

If blocking non-buffered communication is used, we have a deadlock. We need to break cyclic waits but a simple fix here can become cumbersome and buggy in a larger program.

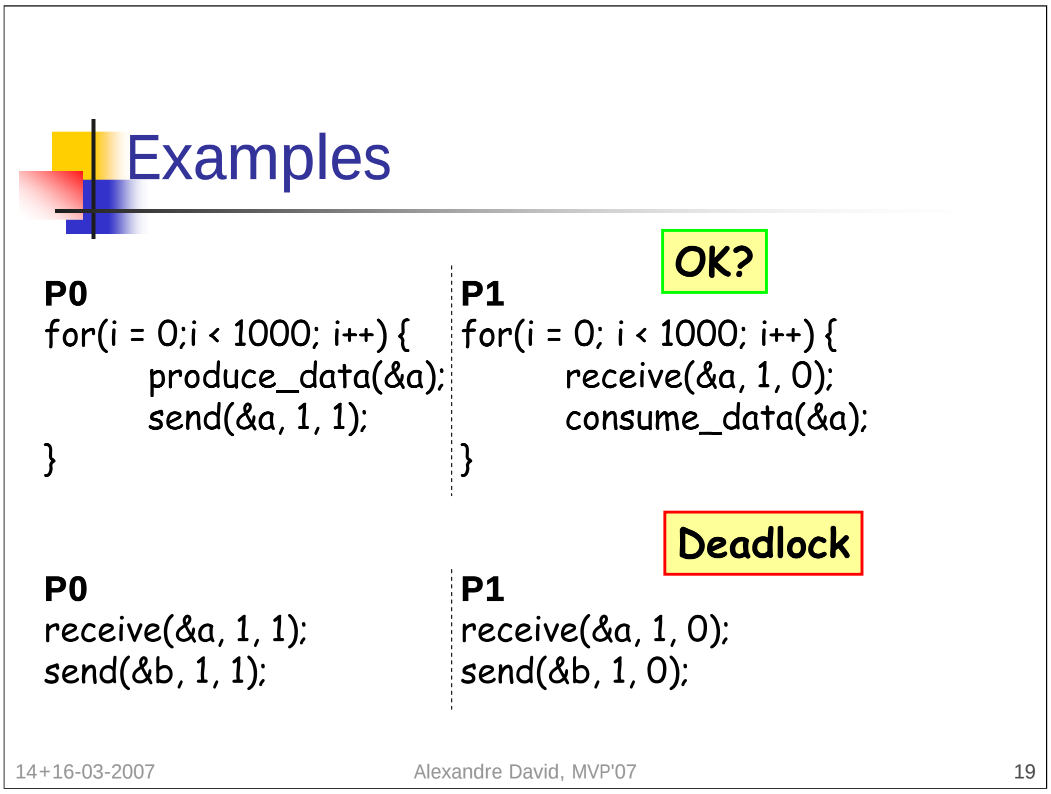


Blocking Buffered Communication



With special hardware.

With special hardware support or without. How to implement a protocol buffering only at the sender's side? Receiver interrupts senders and initiates the transfer when it is ready to receive.



Examples

OK?

P0
```
for(i = 0;i < 1000; i++) {
        produce_data(&a);
        send(&a, 1, 1);
}
```

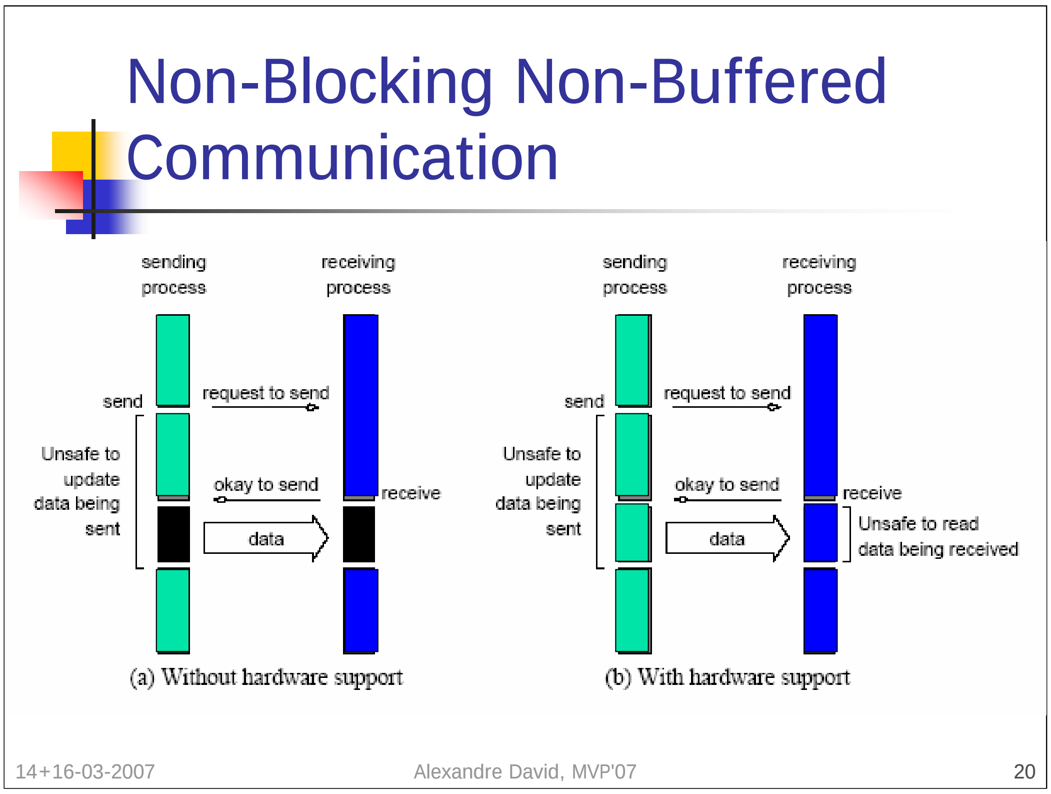
P1
```
for(i = 0; i < 1000; i++) {
        receive(&a, 1, 0);
        consume_data(&a);
}
```

Deadlock

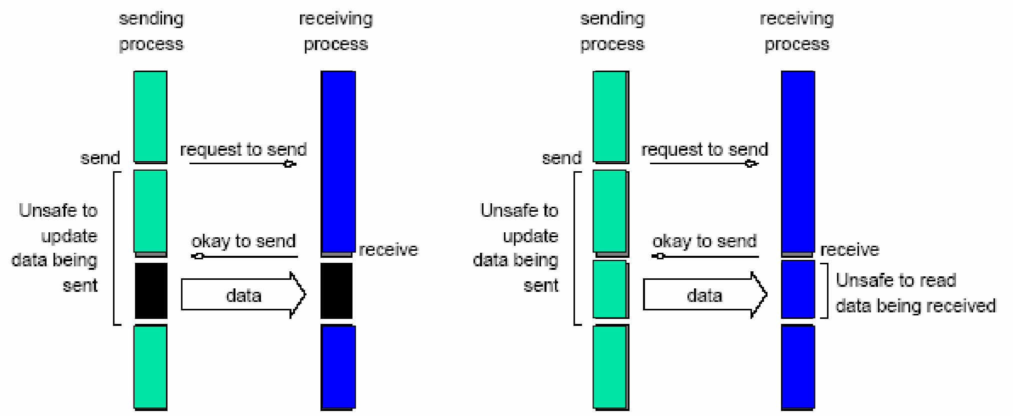
P0
```
receive(&a, 1, 1);
send(&b, 1, 1);
```

P1
```
receive(&a, 1, 0);
send(&b, 1, 0);
```

Almost OK, it depends on the execution speed of the receiver. Buffers are **bounded** and can get filled up easily in this case.



Non-Blocking Non-Buffered Communication



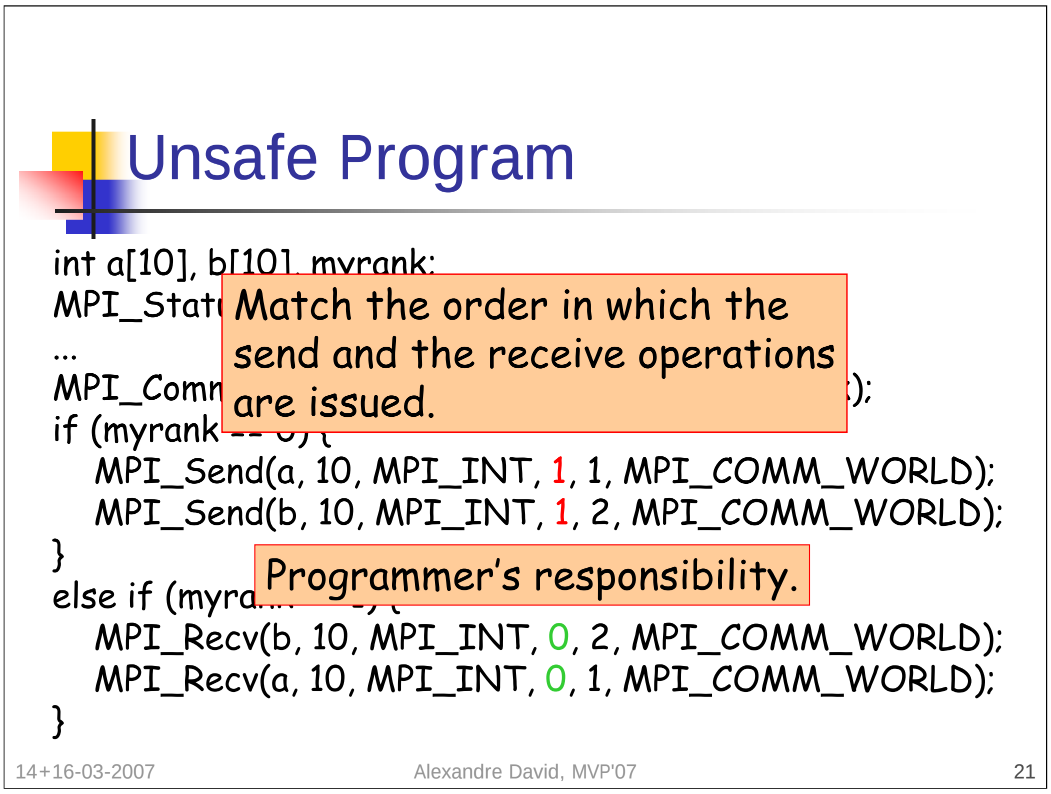
(a) Without hardware support (b) With hardware support

14+16-03-2007 Alexandre David, MVP'07 20

Blocking: Pay semantic correctness in idling (non-buffered) or buffer management (buffered). Here semantic correctness ensured by programmer because the functions (send & receive) return before it is safe to do so. Non-blocking often accompanied by a **check-status** operation.

Note: This is similar to calls to asynchronous I/O operations.

The idling time when the process is waiting for the I/O operation can be used for computation (instead of idling) if the data is not modified (hence possible buffered communication). This may require some program restructuring. Non-blocking communication is further enhanced by **dedicated communication hardware**, such as special network cards with controllers that have direct memory access (DMA).



Unsafe Program

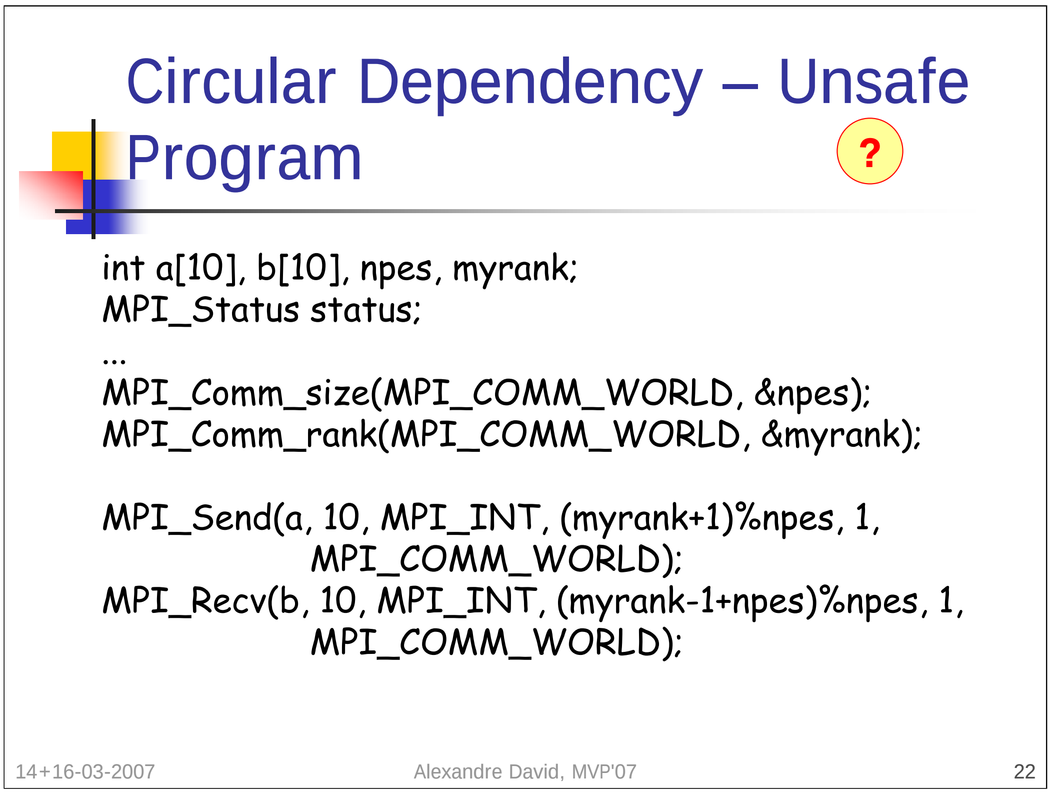
```
int a[10], b[10], myrank;
MPI_Statu
…
MPI_Comm                                       );
if (myrank
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myra
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
```

Match the order in which the send and the receive operations are issued.

Programmer's responsibility.

14+16-03-2007 Alexandre David, MVP'07 21

Different behaviors depending on the implementation of send (with or without buffering, with or without sufficient space). May lead to a **deadlock**.



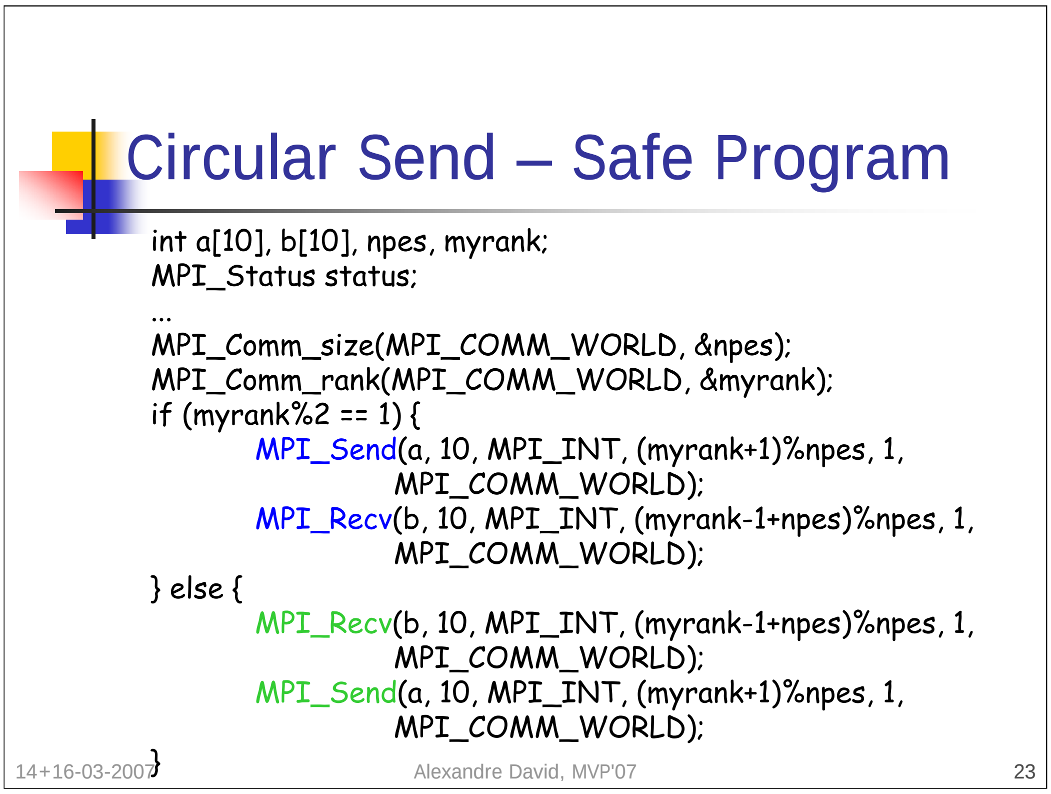
Circular Dependency – Unsafe Program

?

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
                MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                MPI_COMM_WORLD);
```

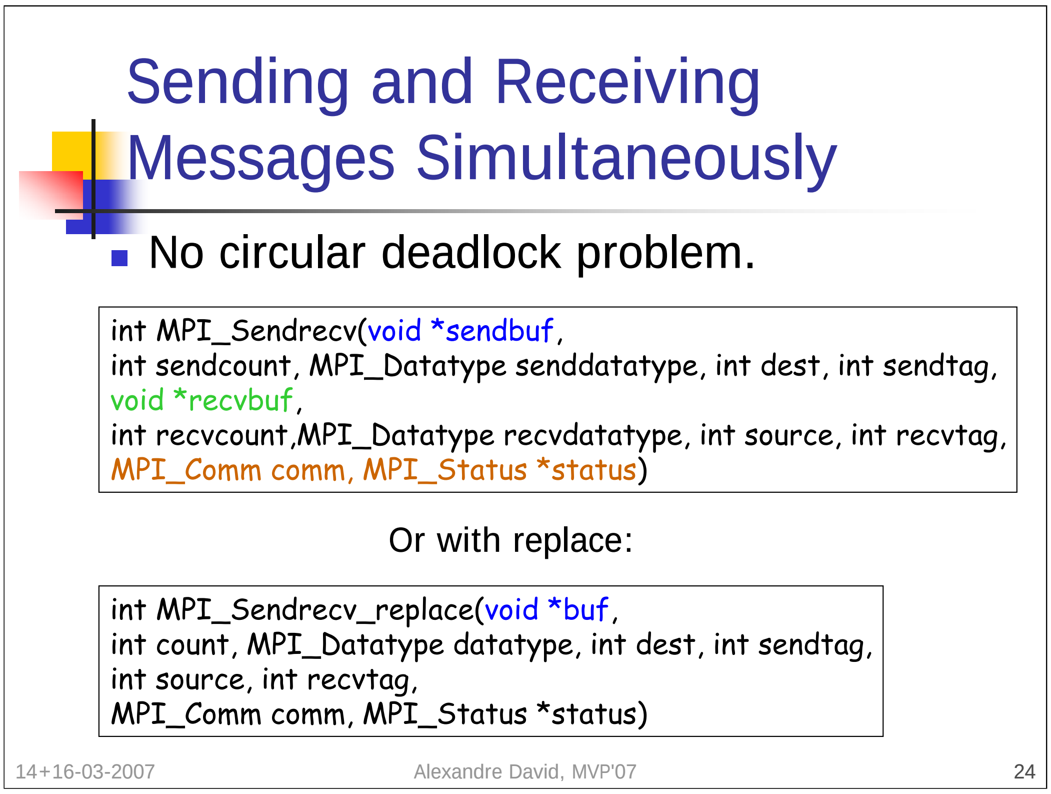
Send messages in a ring. Deadlock if send is blocking.



Circular Send – Safe Program

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
        MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
                  MPI_COMM_WORLD);
        MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                  MPI_COMM_WORLD);
} else {
        MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                  MPI_COMM_WORLD);
        MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
                  MPI_COMM_WORLD);
}
```

Solution similar to the classical dining philosophers problem. Processes are partitioned into two groups: odd and even. Common communication pattern so there is a send & receive function.



Sending and Receiving Messages Simultaneously

- No circular deadlock problem.

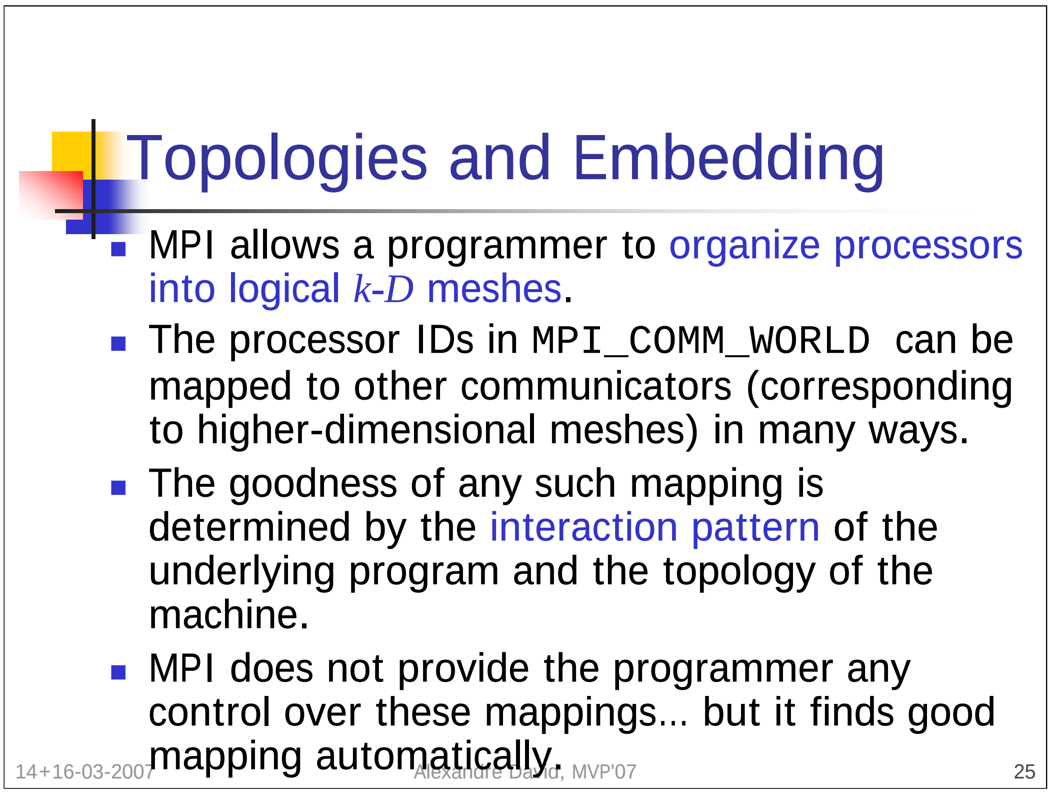
```
int MPI_Sendrecv(void *sendbuf,
int sendcount, MPI_Datatype senddatatype, int dest, int sendtag,
void *recvbuf,
int recvcount,MPI_Datatype recvdatatype, int source, int recvtag,
MPI_Comm comm, MPI_Status *status)
```

Or with replace:

```
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)
```

14+16-03-2007 Alexandre David, MVP'07 24

Exchange of messages. For replace there are constraints on the transferred data type.

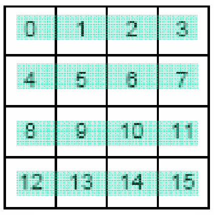


Topologies and Embedding

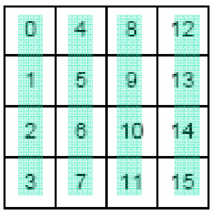
- MPI allows a programmer to organize processors into logical *k-D* meshes.
- The processor IDs in `MPI_COMM_WORLD` can be mapped to other communicators (corresponding to higher-dimensional meshes) in many ways.
- The goodness of any such mapping is determined by the interaction pattern of the underlying program and the topology of the machine.
- MPI does not provide the programmer any control over these mappings... but it finds good mapping automatically.

Mechanism to assign rank to processes does not use any information about the interconnection network, making it impossible to perform topology embeddings in an intelligent manner. Even we had that information, we would have to specify different mappings for different interconnection networks. We want our programs to be **portable**, so let MPI do the job for us, since we know now what is happening underneath.

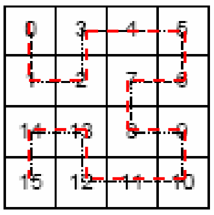
Topologies and Embeddings



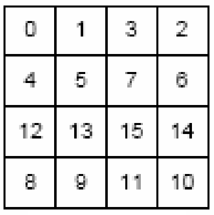
Row-major mapping.



Column-major mapping.

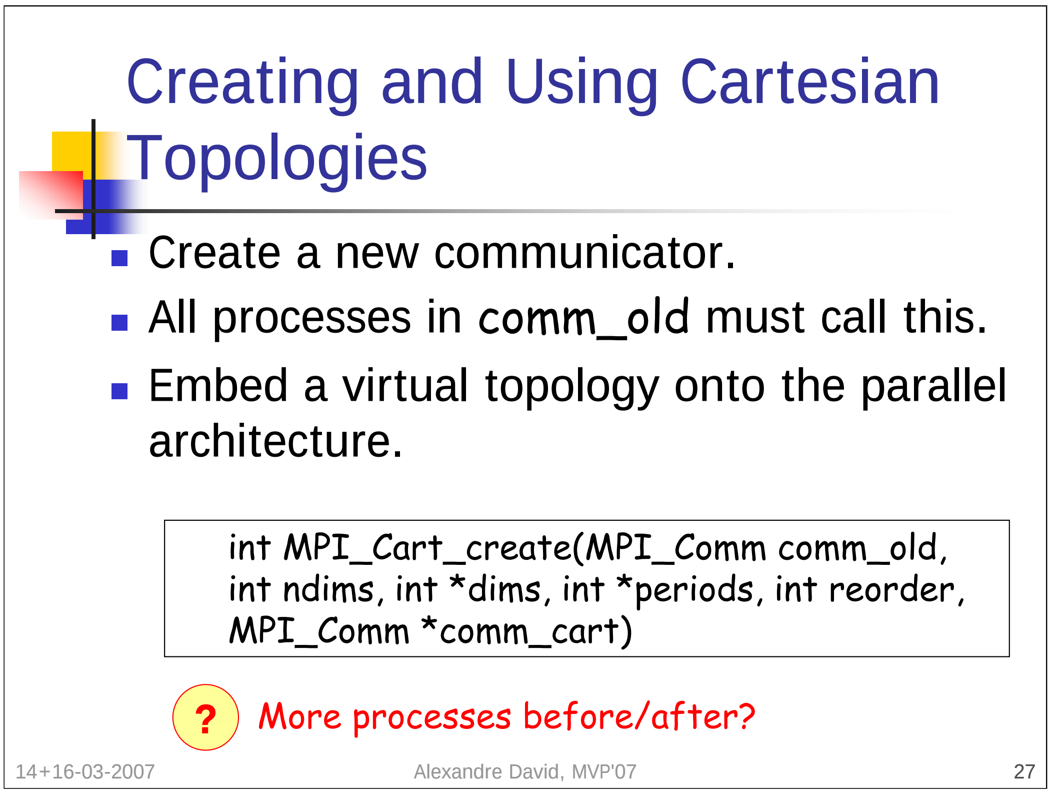


Space-filling curve mapping.



Hypercube mapping.

Recognize Gray code?



Creating and Using Cartesian Topologies

- Create a new communicator.
- All processes in comm_old must call this.
- Embed a virtual topology onto the parallel architecture.

```
int MPI_Cart_create(MPI_Comm comm_old,
int ndims, int *dims, int *periods, int reorder,
MPI_Comm *comm_cart)
```

? More processes before/after?

Multi-dimensional grid topologies.

Arguments:

•ndims: number of dimensions.

•dims[i]: size for every dimension.

•periods[i]: if dim ‘i’ has wrap-around or not.

•reorder: allows to reorder the ranks if that leads to a better embedding.

Notes: For some processes comm_cart may become MPI_COMM_NULL if they are not part of the topology (more processes in comm_old than in the described topology). If the number of processes in the topology is greater than the number of available processes, we have an error.

We can identify processes by a vector = its coordinates in the topology.

Rank-Coordinates Conversion

- Dimensions must match.
- Shift processes on the topology.

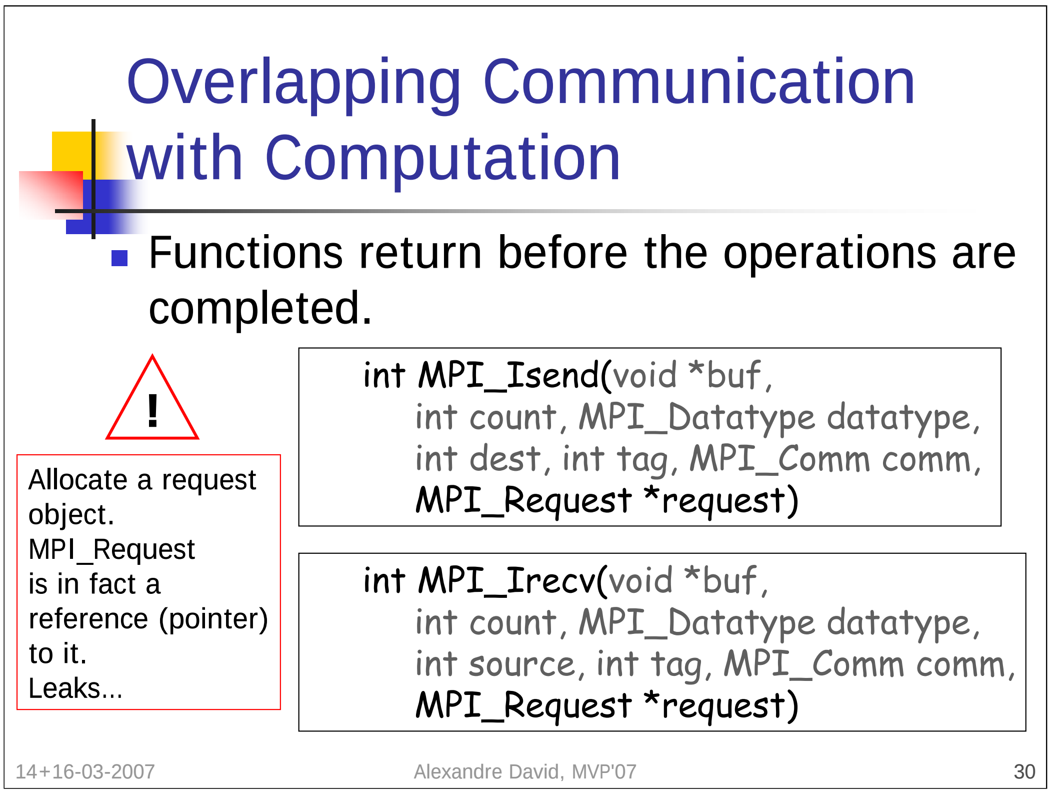
```
int MPI_Cart_coord(MPI_Comm comm_cart,
int rank, int maxdims, int *coords)
```

```
int MPI_Cart_rank(MPI_Comm comm_cart,
int *coords, int *rank)
```

```
int MPI_Cart_shift(MPI_Comm comm_cart,
int dir, int s_step, int *rank_source, int *rank_dest)
```

Overlapping Communication with Computation

- Transmit messages without interrupting the CPU.
- Recall how blocking send/receive operations work.
- Sometimes desirable to have non-blocking.



Overlapping Communication with Computation

- Functions return before the operations are completed.

!

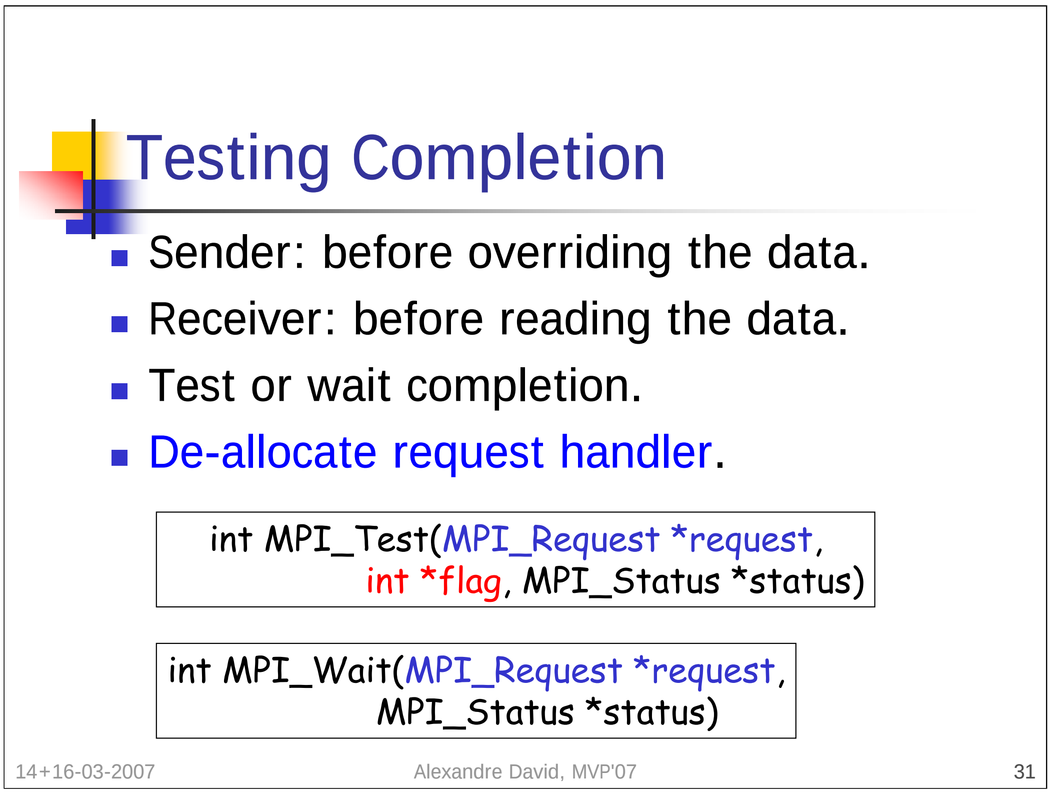
Allocate a request object.
MPI_Request
is in fact a reference (pointer) to it.
Leaks...

```
int MPI_Isend(void *buf,
    int count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm,
    MPI_Request *request)
```

```
int MPI_Irecv(void *buf,
    int count, MPI_Datatype datatype,
    int source, int tag, MPI_Comm comm,
    MPI_Request *request)
```

14+16-03-2007 Alexandre David, MVP'07 30

Later we need to make sure that the operations are completed so the additional 'request' argument provides a handler on the operation for later test.



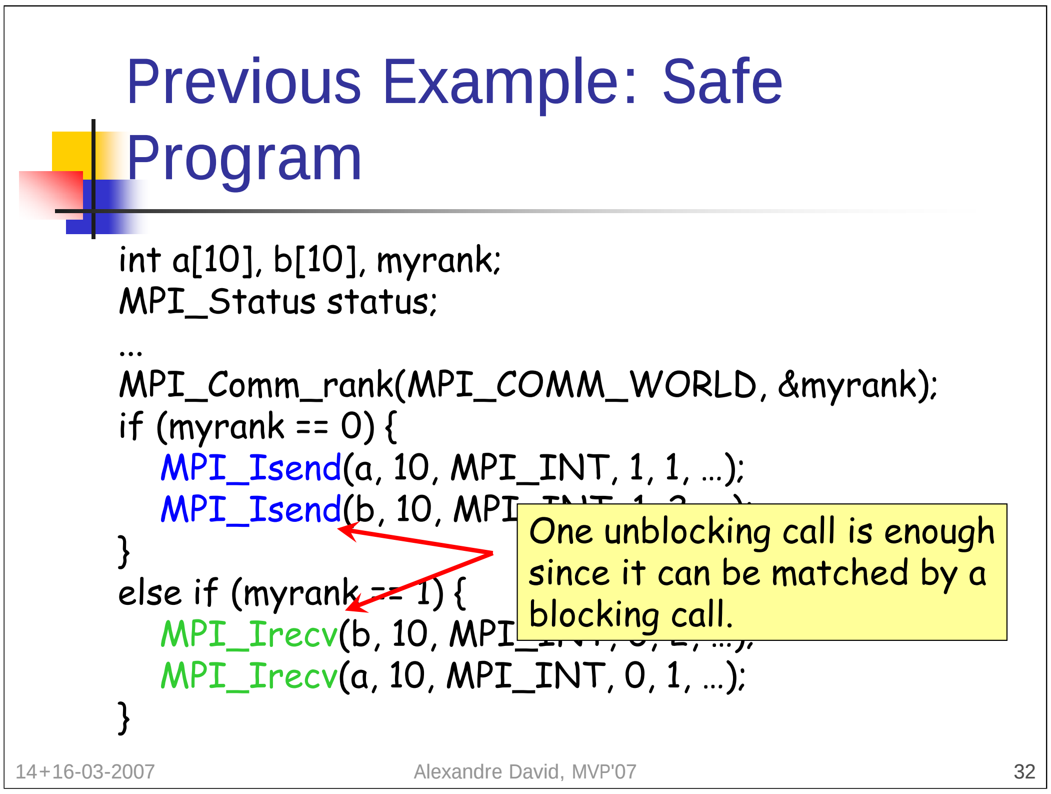
Testing Completion

- Sender: before overriding the data.
- Receiver: before reading the data.
- Test or wait completion.
- De-allocate request handler.

```
int MPI_Test(MPI_Request *request,
             int *flag, MPI_Status *status)
```

```
int MPI_Wait(MPI_Request *request,
             MPI_Status *status)
```

De-allocation **if the blocking operation has finished**. It’s OK to send with non-blocking and receive with blocking.



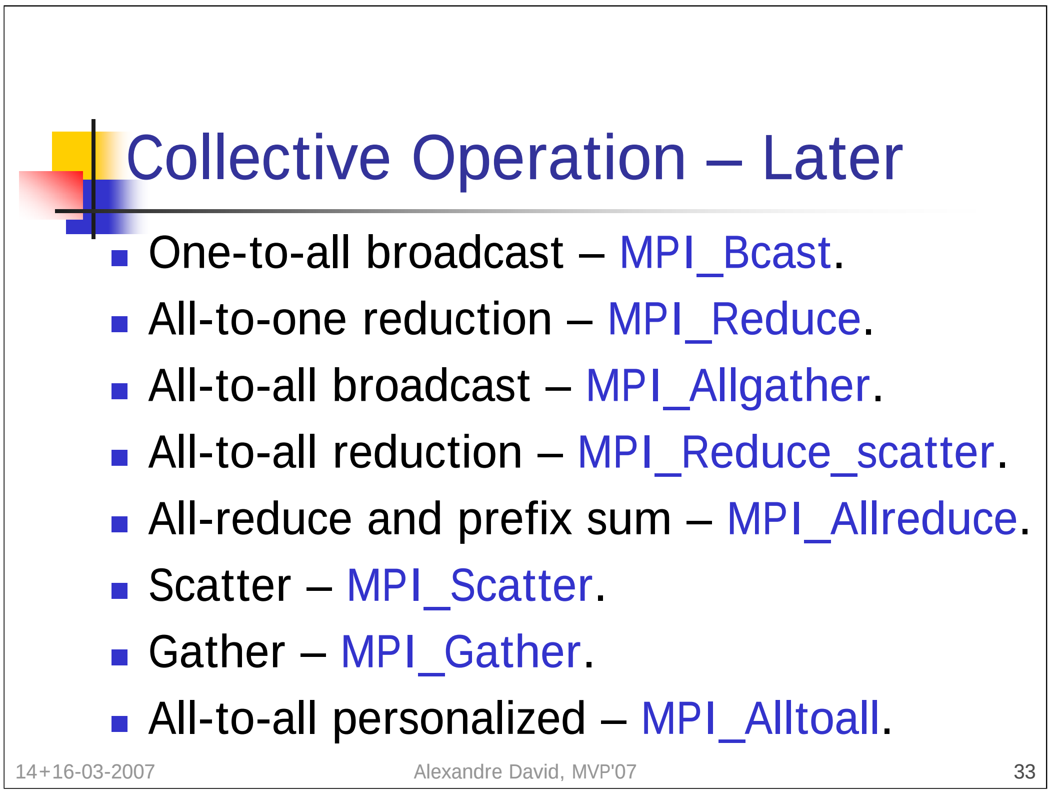
Previous Example: Safe Program

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
   MPI_Isend(a, 10, MPI_INT, 1, 1, …);
   MPI_Isend(b, 10, MPI_INT, 1, 2, …);
}
else if (myrank == 1) {
   MPI_Irecv(b, 10, MPI_INT, 0, 2, …);
   MPI_Irecv(a, 10, MPI_INT, 0, 1, …);
}
```

One unblocking call is enough since it can be matched by a blocking call.

14+16-03-2007 Alexandre David, MVP'07 32

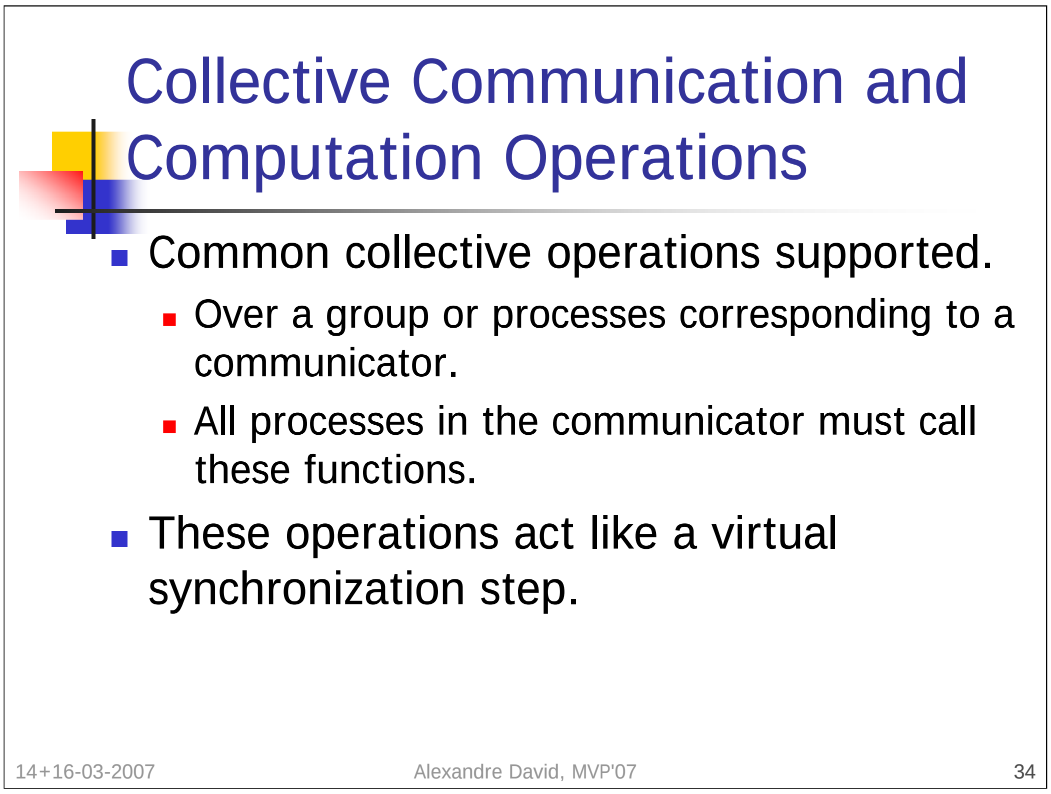
Avoid deadlock. Most of the time, this is at the expense of increased memory usage.



Collective Operation – Later

- One-to-all broadcast – MPI_Bcast.
- All-to-one reduction – MPI_Reduce.
- All-to-all broadcast – MPI_Allgather.
- All-to-all reduction – MPI_Reduce_scatter.
- All-reduce and prefix sum – MPI_Allreduce.
- Scatter – MPI_Scatter.
- Gather – MPI_Gather.
- All-to-all personalized – MPI_Alltoall.

You should know what these operations do.



Collective Communication and Computation Operations

- Common collective operations supported.
  - Over a group or processes corresponding to a communicator.
  - All processes in the communicator must call these functions.
- These operations act like a virtual synchronization step.

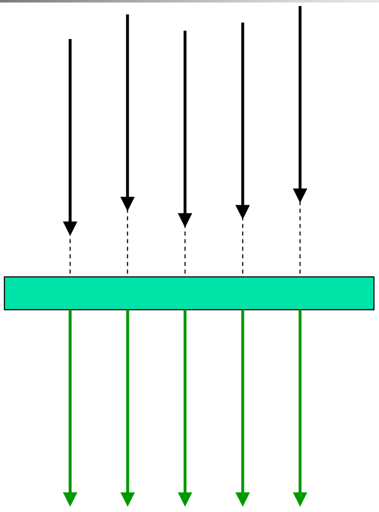
Parallel programs should be written such that they behave correctly even if a global synchronization is performed before and after the collective call.

Barrier

- Communicator: Group of processes that are synchronized.
- The function returns after all processes in the group have called the function.

```
int MPI_Barrier(MPI_Comm comm)
```

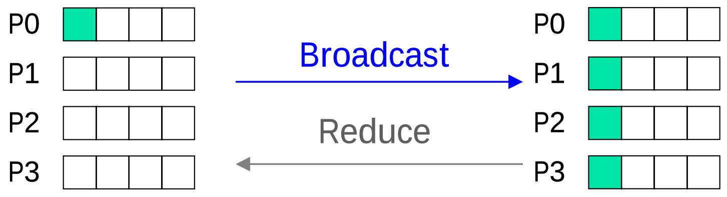
Barrier

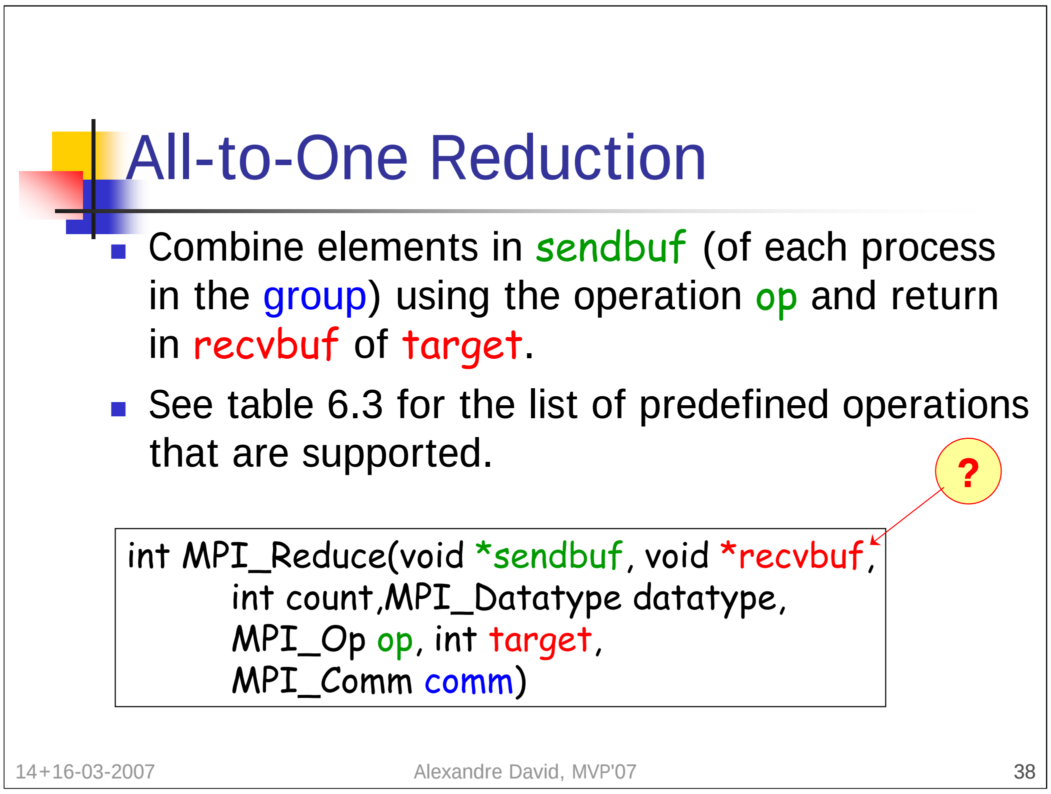


One-to-All Broadcast

- All the processes must call this function, even the receivers.

```
int MPI_Bcast(void *buf,
         int count, MPI_Datatype datatype,
         int source, MPI_Comm comm)
```





All-to-One Reduction

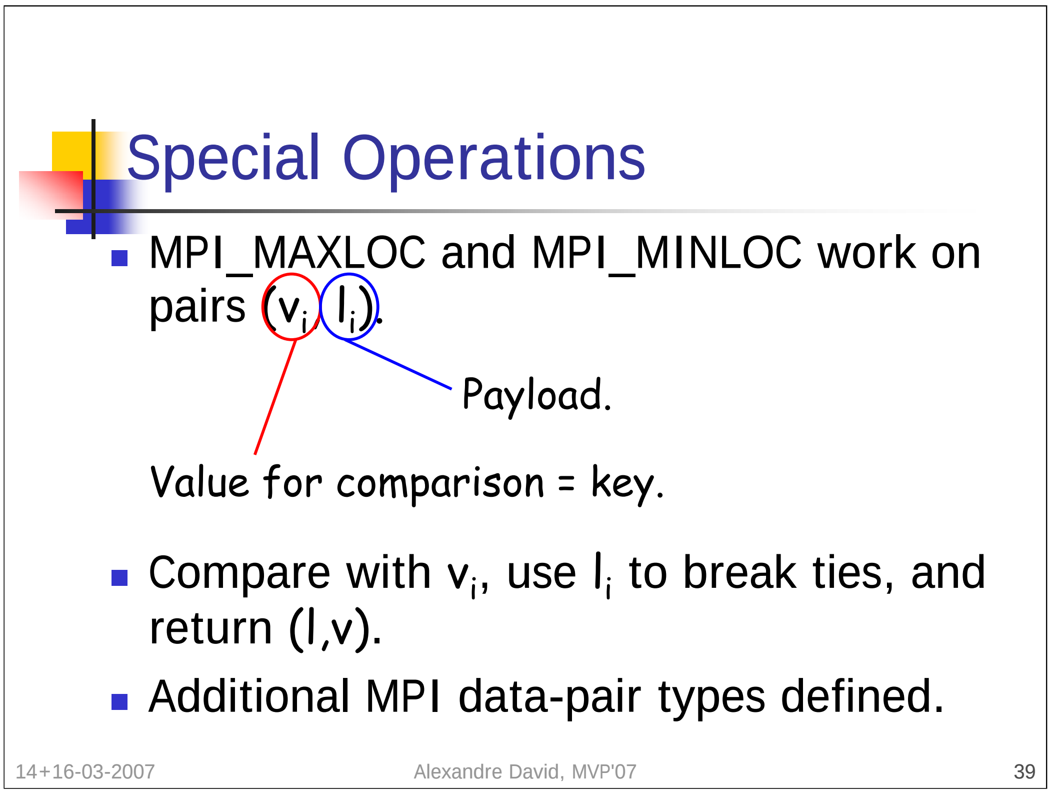
- Combine elements in sendbuf (of each process in the group) using the operation op and return in recvbuf of target.
- See table 6.3 for the list of predefined operations that are supported.

?

```
int MPI_Reduce(void *sendbuf, void *recvbuf,
        int count,MPI_Datatype datatype,
        MPI_Op op, int target,
        MPI_Comm comm)
```

14+16-03-2007 Alexandre David, MVP'07 38

Constraint on the count of items of type datatype. All the processes call this function even those that are not the target and they all provide a recvbuf. When count > 1, the operation is applied element-wise. **Why do they all need a recvbuf?**



Special Operations

- MPI_MAXLOC and MPI_MINLOC work on pairs (v_i, l_i).

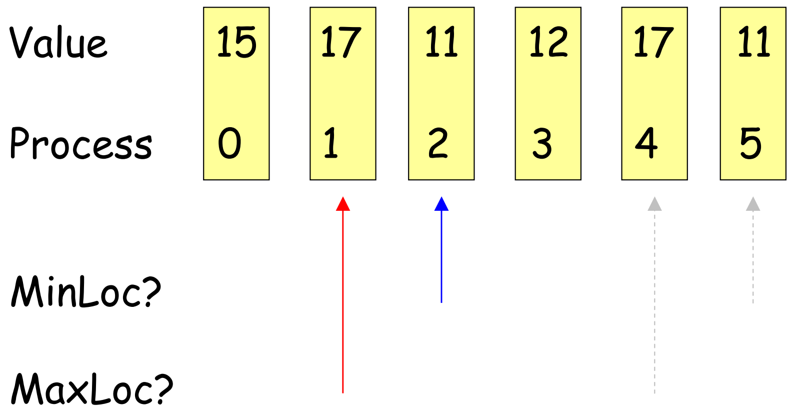
Payload.

Value for comparison = key.

- Compare with v_i, use l_i to break ties, and return (l,v).
- Additional MPI data-pair types defined.

See table 6.4 for the different pair data types.

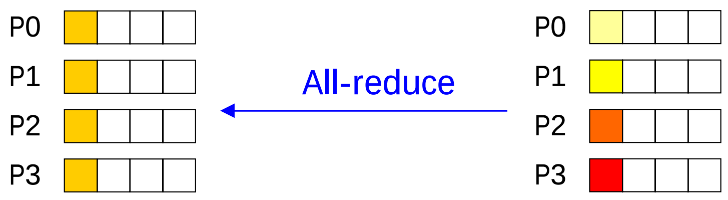
Example



All-Reduce

- No target argument since all processes receive the result.

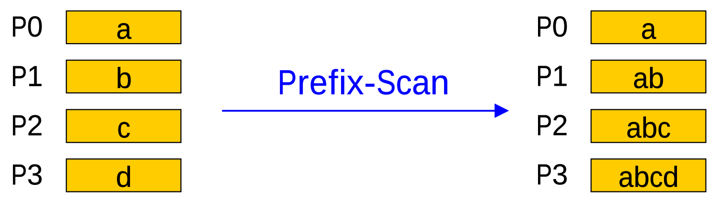
```
int MPI_Allreduce(void *sendbuf, void *recvbuf,
int count, MPI_Datatype datatype,
MPI_Op op, MPI_Comm comm)
```



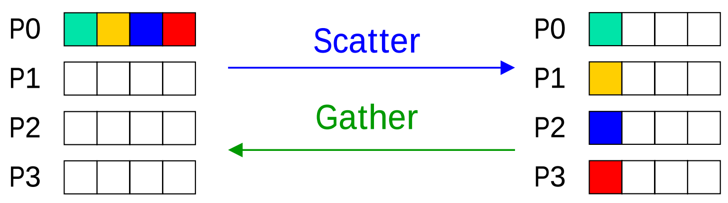
Prefix-Operations

- Not only sums.
- Process *j* has prefix s_j as expected.

```
int MPI_Scan(void *sendbuf, void *recvbuf,
        int count, MPI_Datatype datatype, MPI_Op op,
        MPI_Comm comm)
```

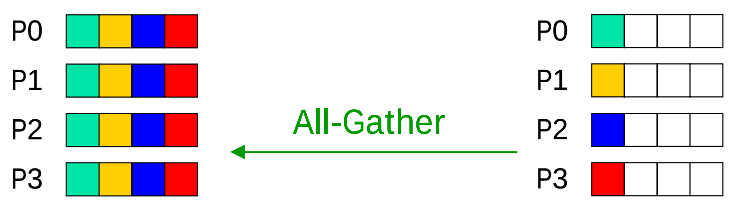


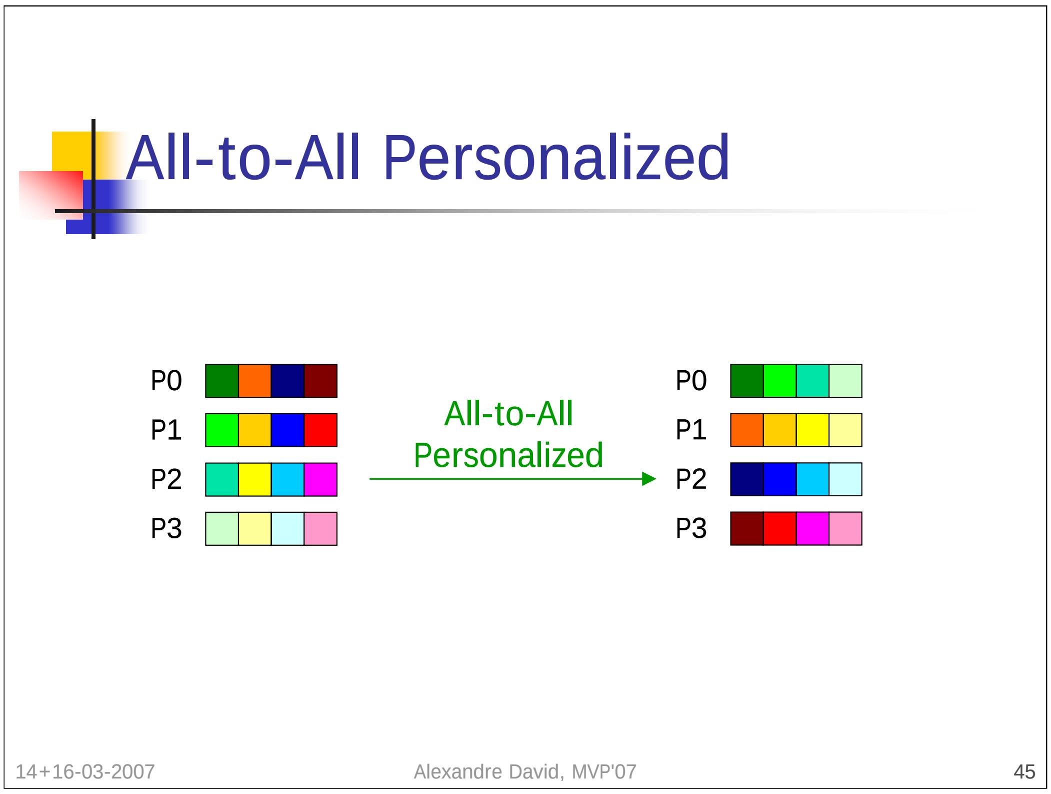
Scatter and Gather



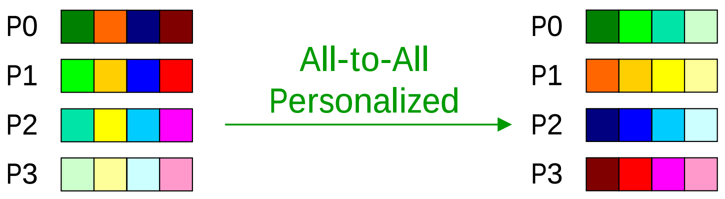
All-Gather

- Variant of gather.

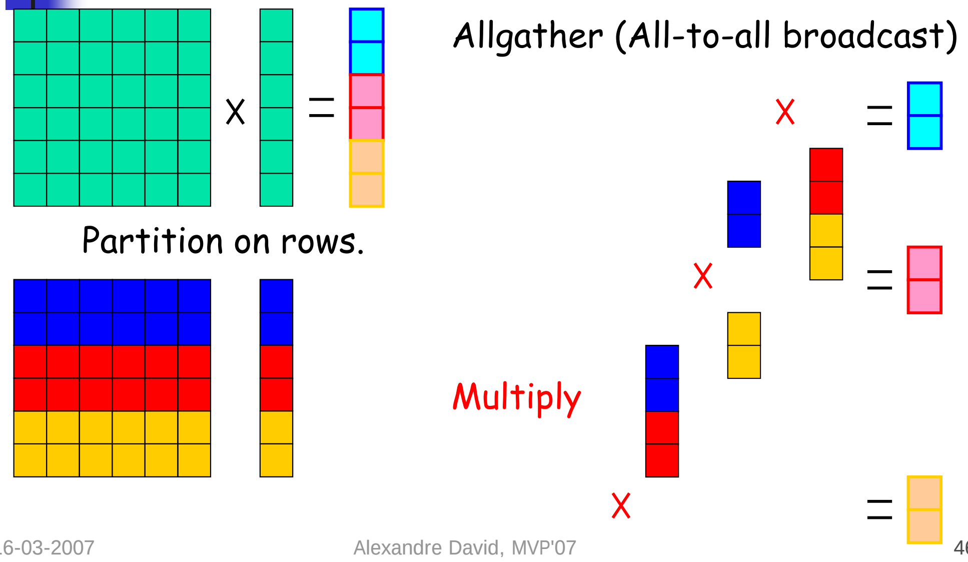


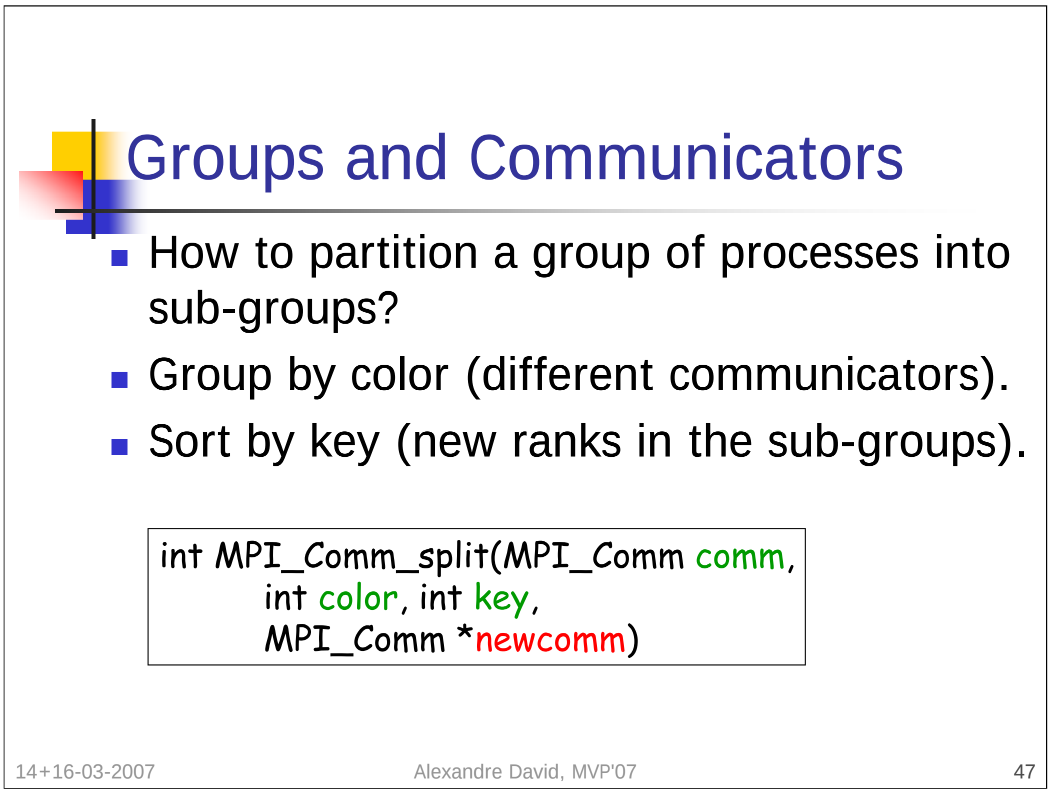


All-to-All Personalized



Example Matrix*Vector (Program 6.4)





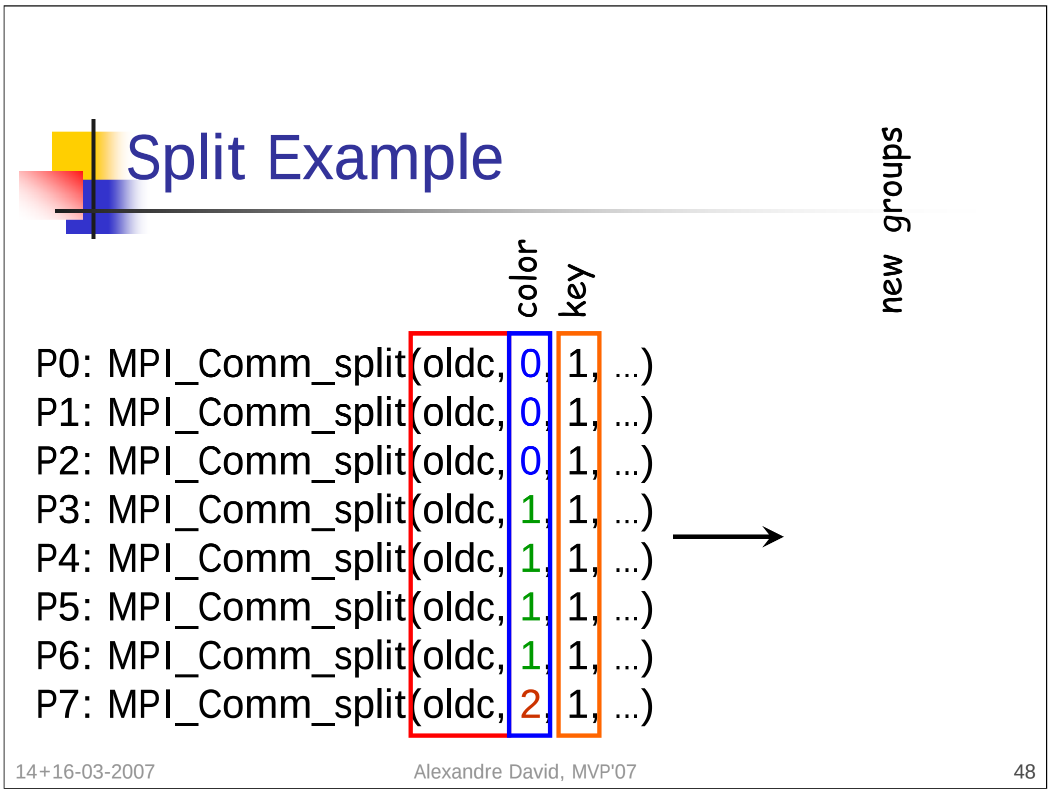
Groups and Communicators

- How to partition a group of processes into sub-groups?
- Group by color (different communicators).
- Sort by key (new ranks in the sub-groups).

```
int MPI_Comm_split(MPI_Comm comm,
        int color, int key,
        MPI_Comm *newcomm)
```

14+16-03-2007 Alexandre David, MVP'07 47

Sometimes parallel algorithms need a restricted communication to certain subsets of processes.



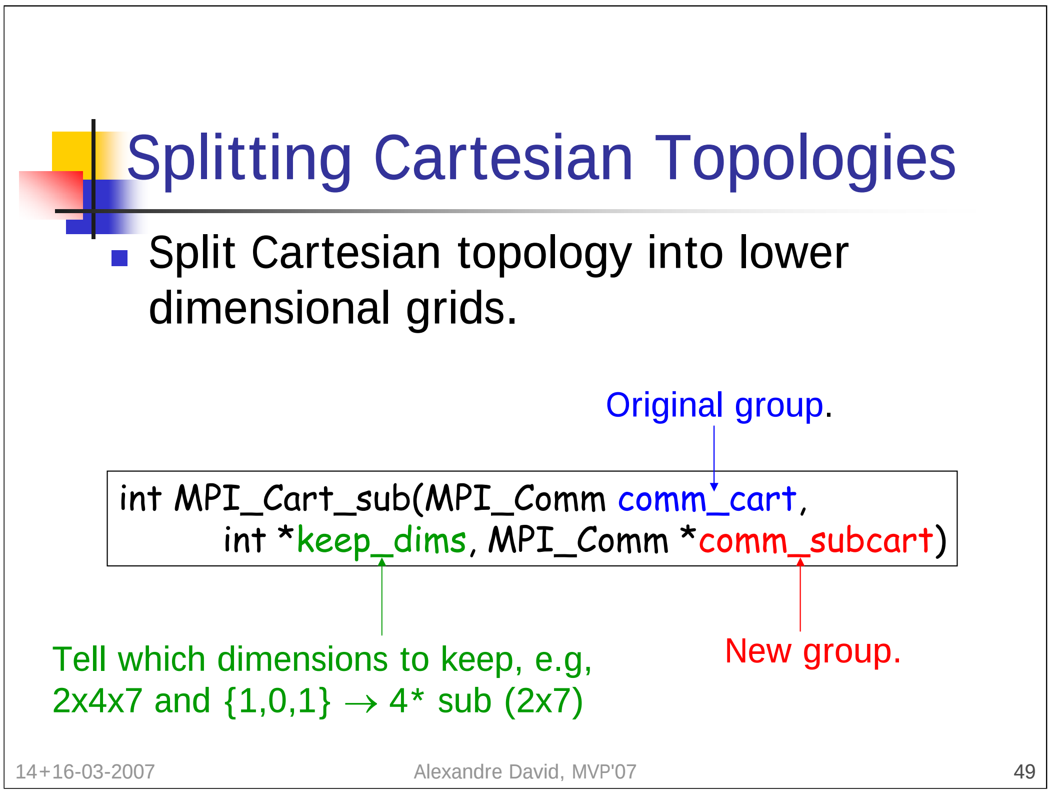
Split Example

color key new groups

```
P0: MPI_Comm_split(oldc, 0, 1, ...)
P1: MPI_Comm_split(oldc, 0, 1, ...)
P2: MPI_Comm_split(oldc, 0, 1, ...)
P3: MPI_Comm_split(oldc, 1, 1, ...)
P4: MPI_Comm_split(oldc, 1, 1, ...)
P5: MPI_Comm_split(oldc, 1, 1, ...)
P6: MPI_Comm_split(oldc, 1, 1, ...)
P7: MPI_Comm_split(oldc, 2, 1, ...)
```

$\longrightarrow$

14+16-03-2007 Alexandre David, MVP'07 48



Splitting Cartesian Topologies

- Split Cartesian topology into lower dimensional grids.

Original group.

```
int MPI_Cart_sub(MPI_Comm comm_cart,
        int *keep_dims, MPI_Comm *comm_subcart)
```

Tell which dimensions to keep, e.g, 2x4x7 and {1,0,1} $\rightarrow$ 4* sub (2x7)

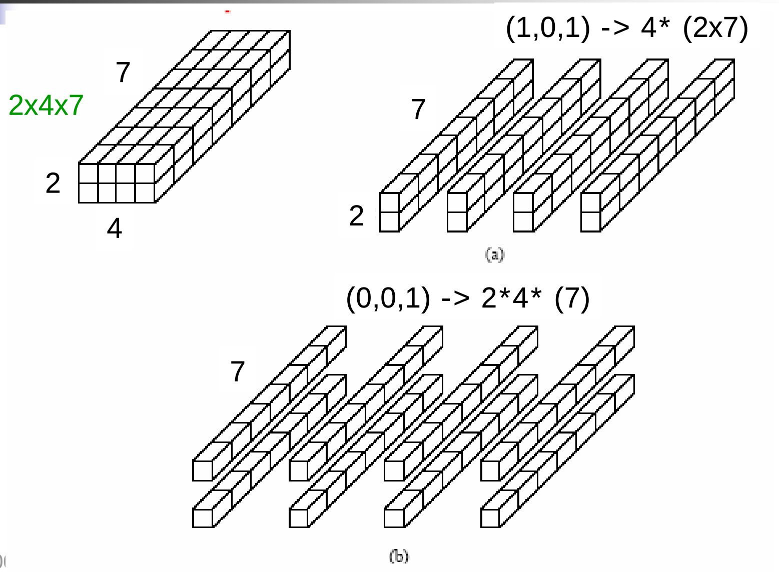
New group.

14+16-03-2007 Alexandre David, MVP'07 49

The keep_dims (boolean) array tells which dimensions to keep for the new sub-group partitioning. The coordinate will match, e.g., (1,2,3) in the original will give (1,3) and will be in the 2nd sub-group.

Example

original 2x4x7

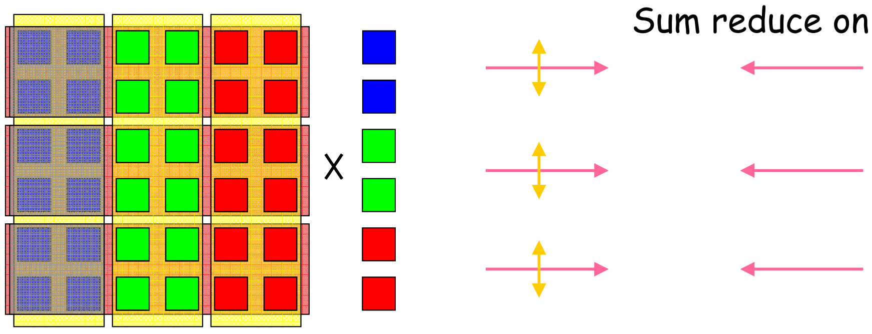


Example Matrix*Vector (Program 6.8)

Partition.

Distribute vector.

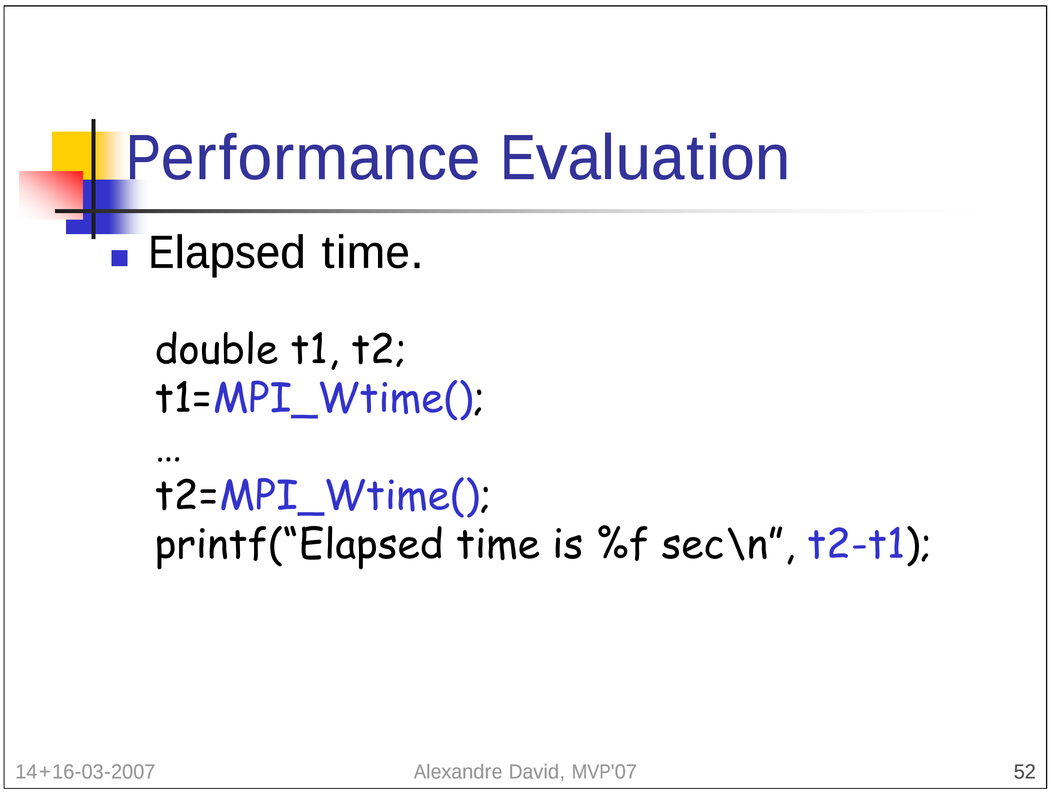
Sum reduce on rows.



Row sub-topology.

Colum sub-topology.

Local multiplication.



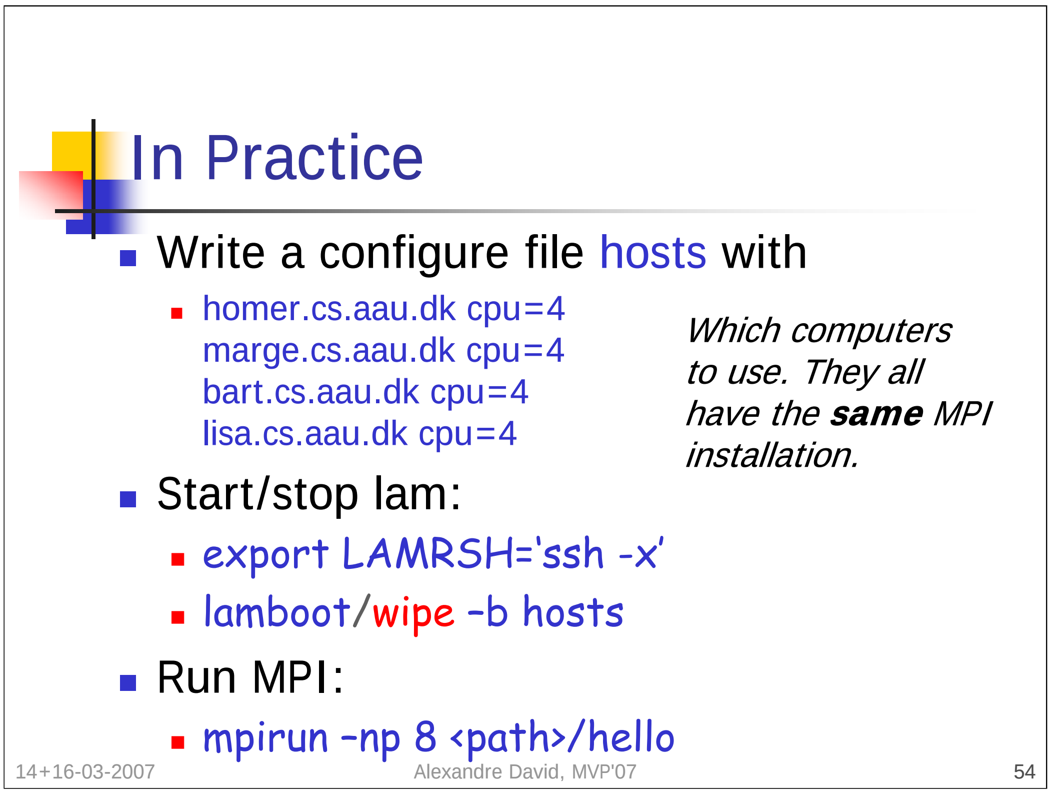
Performance Evaluation

- Elapsed time.

```
double t1, t2;
t1=MPI_Wtime();
…
t2=MPI_Wtime();
printf("Elapsed time is %f sec\n", t2-t1);
```

Howto

- Compile a hello.c MPI program:
  - `mpicc -Wall -O2 -o hello hello.c`
- Start Lam:
  - `lamboot`
- Run:
  - `mpirun -np 4 ./hello`
- Clean-up before logging off:
  - `wipe`



In Practice

- Write a configure file hosts with
  - homer.cs.aau.dk cpu=4
    marge.cs.aau.dk cpu=4
    bart.cs.aau.dk cpu=4
    lisa.cs.aau.dk cpu=4

*Which computers to use. They all have the **same** MPI installation.*

- Start/stop lam:
  - export LAMRSH='ssh -x'
  - lamboot/wipe -b hosts
- Run MPI:
  - mpirun -np 8 <path>/hello

14+16-03-2007 Alexandre David, MVP'07 54

There are different implementations of MPI. LAM/MPI is a bit old, OpenMPI is more recent. Depending on the vendor you can have something else.