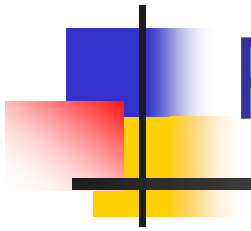
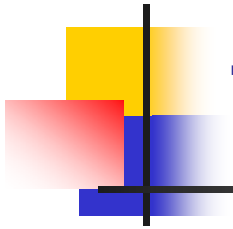


Programming Using MPI

Alexandre David

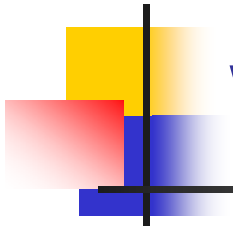
1.2.05

adavid@cs.aau.dk



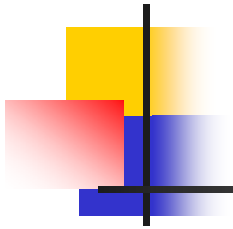
Topic Overview

- 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



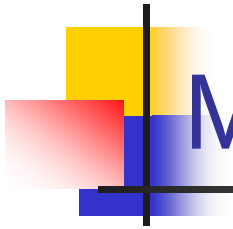
Why MPI?

- 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.
  - Private address space → no mutex.



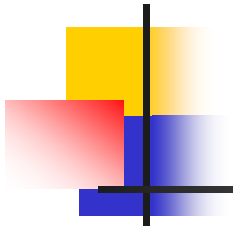
MPI Difficulties

- Many low-level details left to the programmer.
- Specify both sending & receiving messages.
- Sending & receiving must match (order, length, types).
- Efficient non-blocking communication more difficult to use because of many assumptions.
- Deadlocks & livelocks.
- Distributed algorithms with more complex communication protocols.



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.

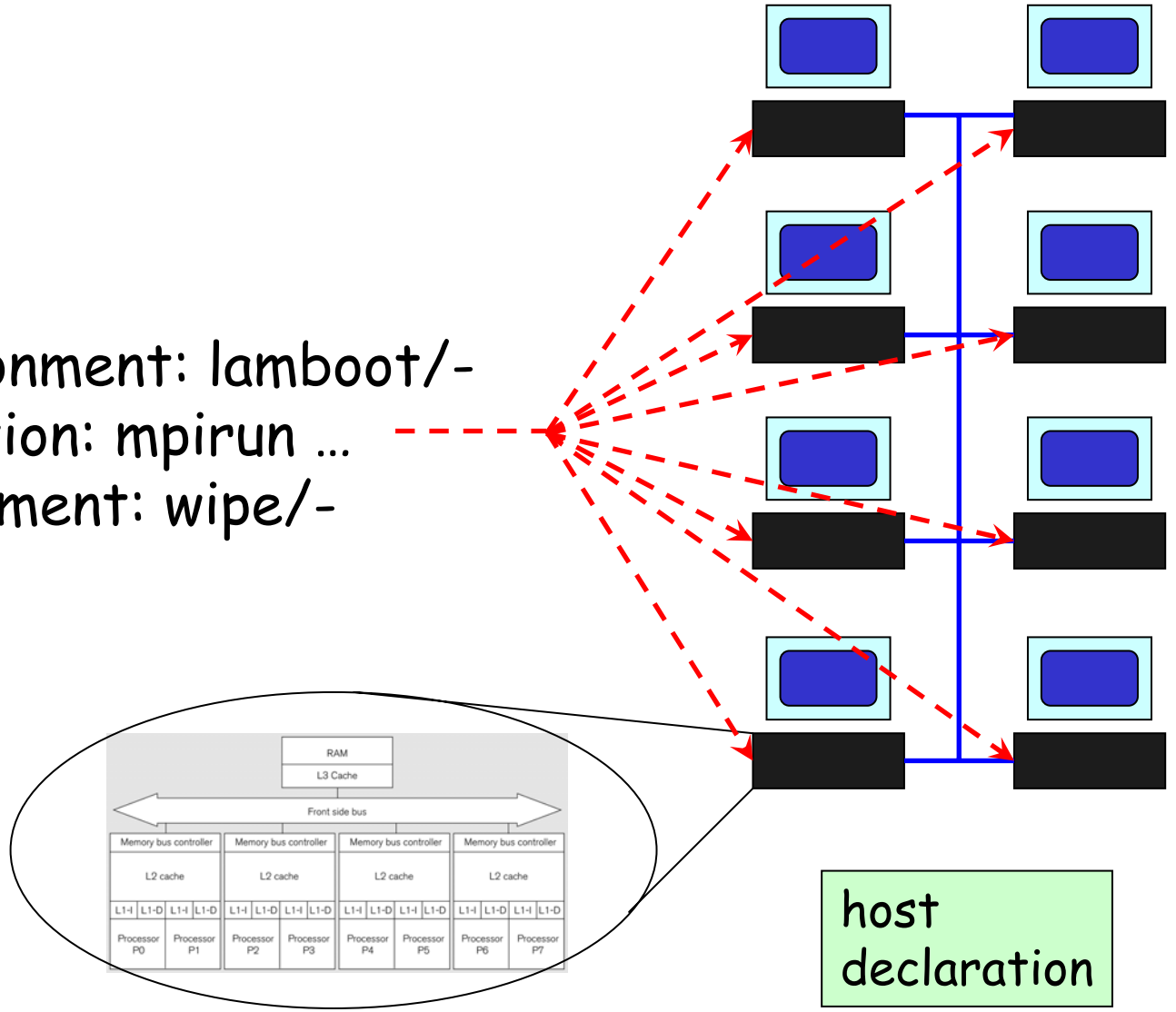


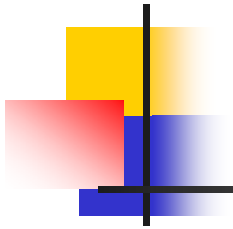
MPI Basics

- Global view “language” (library in fact).
- Only way to communicate is by sending messages.
- Private address space.
- Number of processes is fixed.
- Process scheduling external to MPI.
- Run the same program on different machines.

MPI – The Big Picture

start the environment: lamboot/-
run the application: mpirun …
end the environment: wipe/-

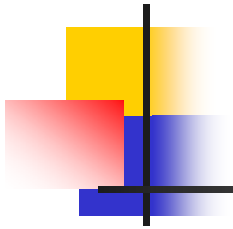




MPI Features

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

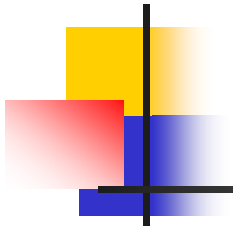
```
send(const void *sendbuf, int nelem, int dest)
receive(void *recvbuf, int nelem, int src)
```



Six Golden MPI Functions

- Total ~125 functions.
- 6 most used function.

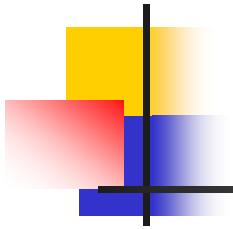
MPI_Init	Initializes MPI.
MPI_Finalize	Terminates MPI.
MPI_Comm_size	Determines the number of processes.
MPI_Comm_rank	Determines the label of the calling process.
MPI_Send	Sends a message.
MPI_Recv	Receives a message.



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()
```



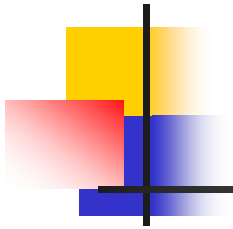
MPI Functions: Communicator

- 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)
```

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;
}
```

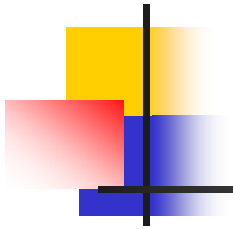


MPI Functions: Send, Recv

- 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)
```

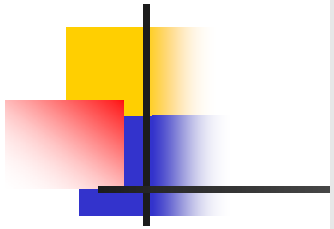
```
typedef struct MPI_Status {
        int MPI_SOURCE;
        int MPI_TAG;
        int MPI_ERROR;
};
```



Length of Received Message

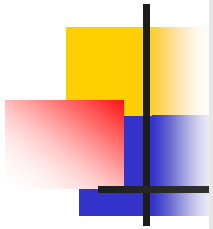
- Not directly accessible.
- Reminder: The returned int says if the call was successful or not.

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



Count3s

```
 1  #include <stdio.h>
 2  #include "mpi.h"
 3  #include "globals.h"
 4
 5  int main(argc, argv)
 6  int argc;
 7  char **argv;
 8  {
 9    int myID, value, numProcs;
10    MPI_Status status;
11
12    MPI_Init(&argc, &argv);
13    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
14    MPI_Comm_rank(MPI_COMM_WORLD, &myID);
15
16    length_per_process=length/numProcs;
17    myArray=(int *) malloc(length_per_process*sizeof(int));
18
19    /* Read the data, distribute it among the various processes */
20    if(myID==RootProcess)
21    {
22      if((fp=fopen(*argv, "r"))==NULL )
23      {
24        printf("fopen failed on %s\n", filename);
25        exit(0);
26      }
27      fscanf(fp,"%d", &length);      /* read input size */
28
29      for(p=0; p<numProcs-1; p++)    /* read data on behalf of each */
30      {                              /* of the other processes */
31        for(i=0; i<length_per_process; i++)
32        {
33          fscanf(fp,"%d", myArray+i);
34        }
35        MPI_Send(myArray, length_per_process, MPI_INT, p+1,
36                 tag, MPI_COMM_WORLD);
```

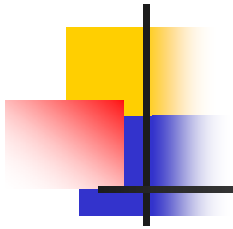


Count3s

```
37        }
38
39        for(i=0; i<length_per_process; i++)  /* Now read my data */
40        {
41          fscanf(fp,"%d", myArray+i);
42        }
43      }
44      else
45      {
46        MPI_Recv(myArray, length_per_process, MPI_INT, RootProcess,
47                 tag, MPI_COMM_WORLD, &status);
48      }
49
50      /* Do the actual work */
51      for(i=0; i<length_per_process; i++)
52      {
53        if(myArray[i]==3)
54        {
55          myCount++;                          /* Update local count */
56        }
57      }
58
59      MPI_Reduce(&myCount,&globalCount, 1, MPI_INT, MPI_SUM,
60                 RootProcess, MPI_COMM_WORLD);
61
62      if(myID==RootProcess)
63      {
64        printf("Number of 3's: %d\n", globalCount);
65      }
66      MPI_Finalize()
67      return 0;
68    }
```

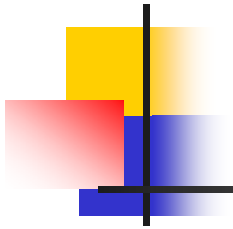
```
16  length_per_process=length/size;
17  myArray=(int *) malloc(length_per_process*sizeof(int));
18
19  array=(int *) malloc(length*sizeof(int));
20
21  /* Read the data, distribute it among the various processes */
22  if(myID==RootProcess)
23  {
24    if((fp=fopen(*argv, "r"))==NULL)
25    {
26      printf("fopen failed on %s\n", filename);
27      exit(0);
28    }
29    fscanf(fp,"%d", &length);        /* read input size */
30
31                                                    ut file */
32
33
34    }
35  }
36
37  MPI_Scatter(Array, length_per_process, MPI_INT,
38              myArray, length_per_process, MPI_INT,
39              RootProcess, MPI_COMM_WORLD);
```

Improvement with MPI_Scatter:
collective MPI communication primitive.



MPI Functions: Data Types

- MPI_Datatype.
- Correspondence MPI $\leftrightarrow$ C data types.
- MPI_BYTE and MPI_PACKED MPI specifics.
- MPI_{CHAR,SHORT,INT,LONG,FLOAT,DOUBLE...} see mpi.h.



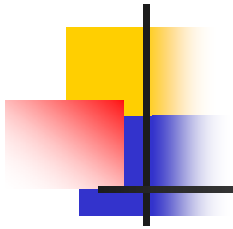
Principles of Message-Passing Programming

Minimize interactions.
Local accesses.

- 2 key attributes:
  - partitioned address space &
  - only explicit parallelization.

Expensive but costs are explicit.

- 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.



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.

Send/Recv Example

P0

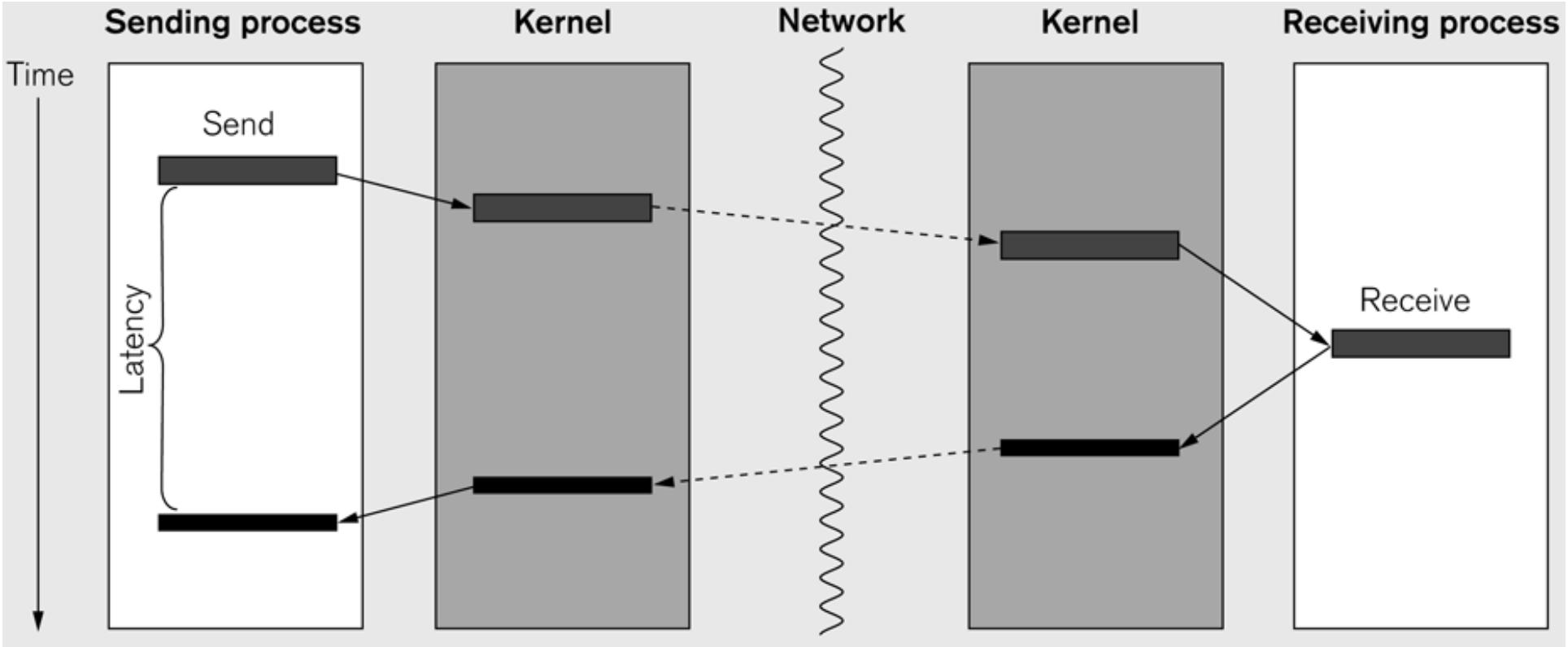
```
a=100;
send(&a, 1, 1);
a=0;
```

P1

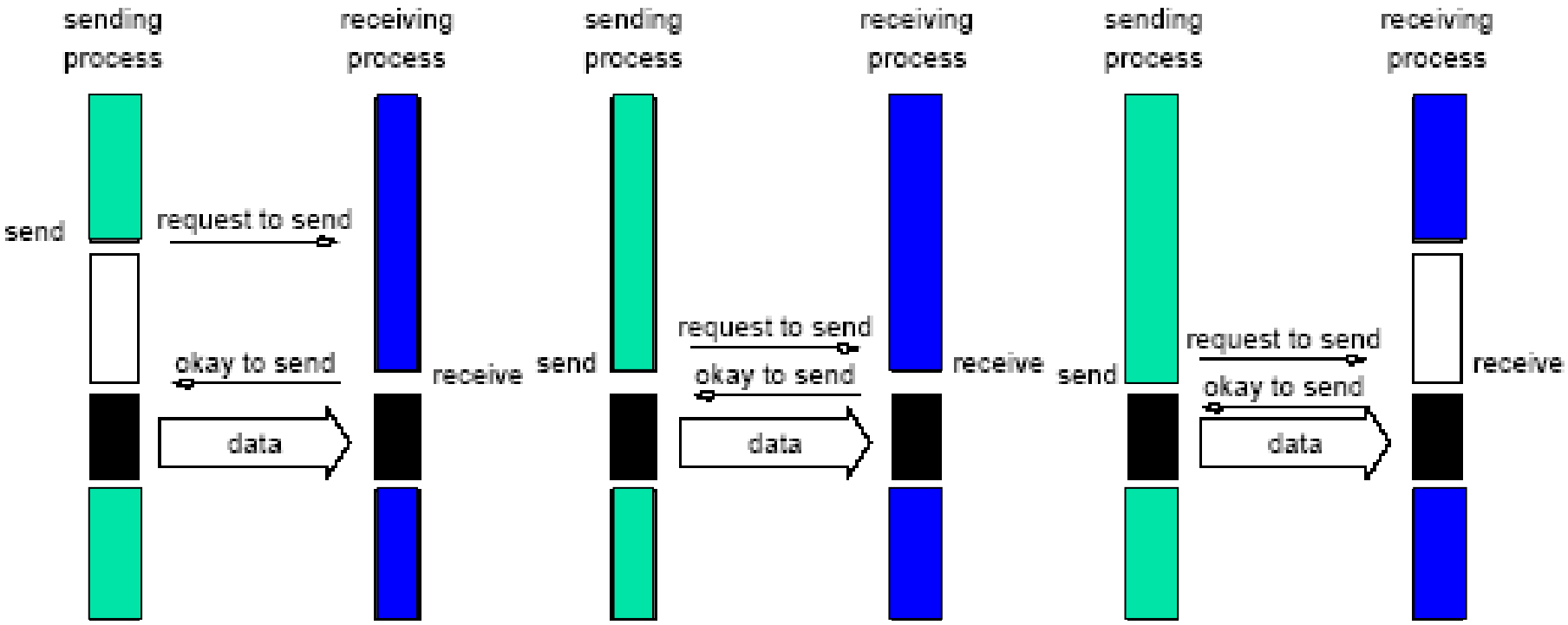
```
receive(&a, 1, 0);
printf("%d\n",a);
```

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

Latency


Sending process
Kernel
Network
Kernel
Receiving process
Time
Send
Latency
Receive

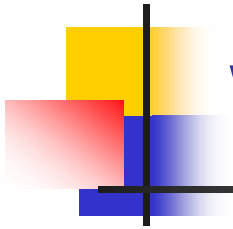
Blocking Non-Buffered Communication



(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



What Happens There?

Simple exchange of 'a'?

?

P0

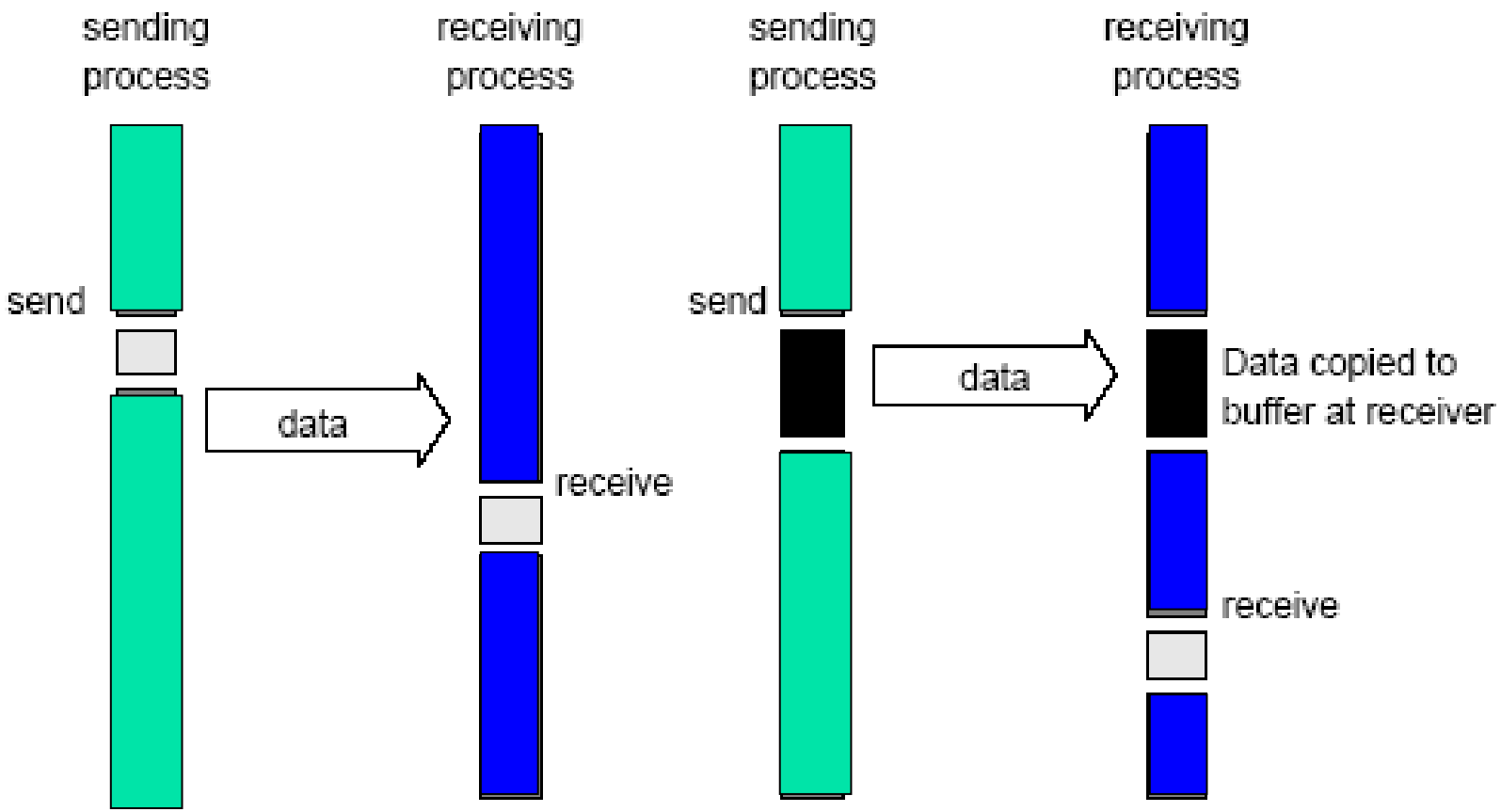
```
send(&a, 1, 1);
recv(&b, 1, 1);
```

P1

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

Blocking Buffered Communication

MPI_Bsend/MPI_Brecv



With special hardware.

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

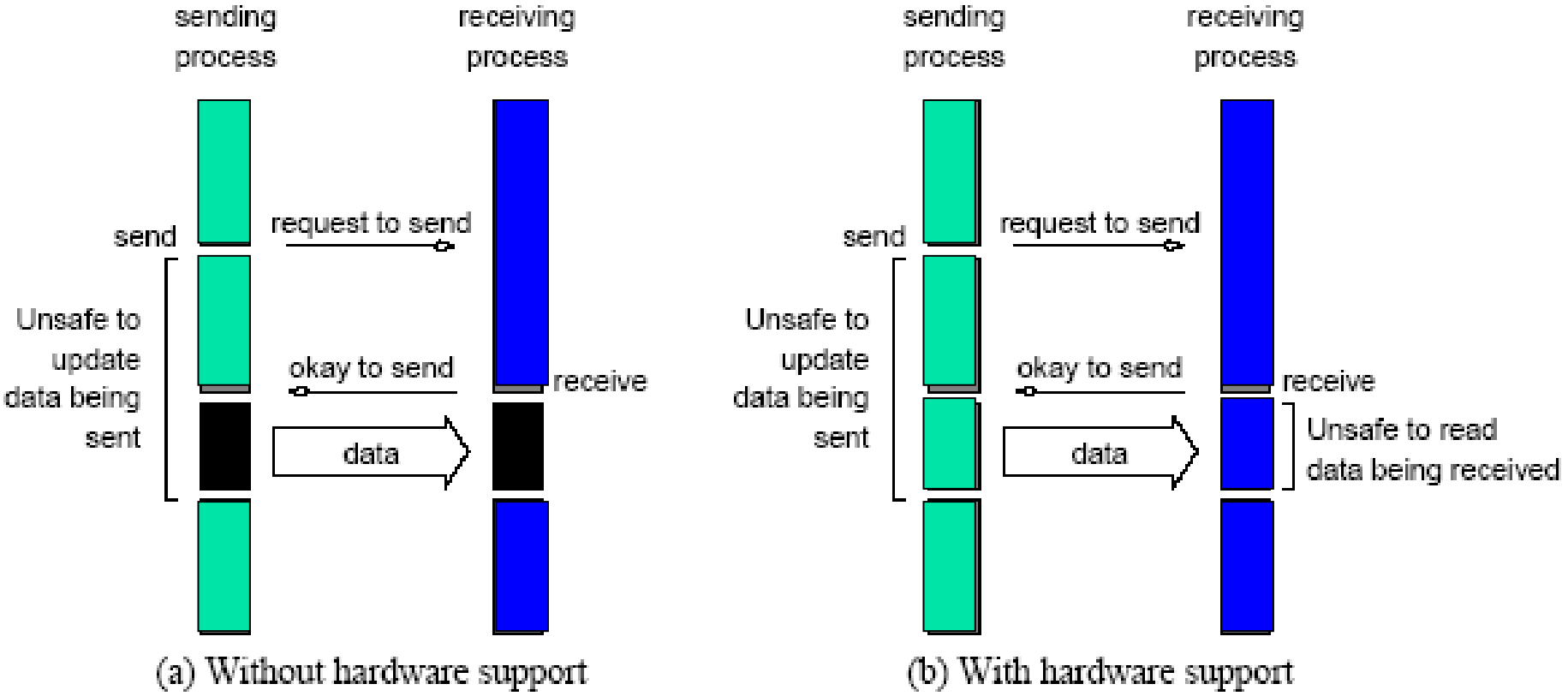
```
receive(&a, 1, 1);
send(&b, 1, 1);
```

P1

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

Non-Blocking Non-Buffered Communication

MPI_Ibsend/MPI_Ibrecv

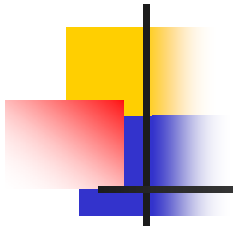


Unsafe Program

```
int a[10], b[10], myrank;
MPI_Stat
...
MPI_Comm                                    );
if (myrank == 0) {
    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.

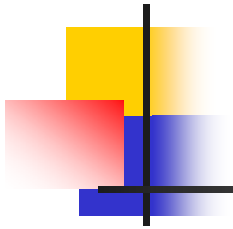
Programmer's responsibility.



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);
```



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);
}
```

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)
```

Creating Groups

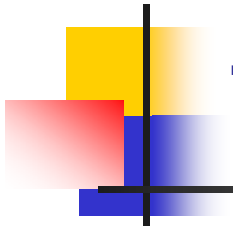
- Create sub-groups by selecting processes from the current group.

```
int MPI_Group_incl(MPI_Group group,
                   int size, int *ranks,
                   MPI_Group *newGroup)
```

matrix of ranks,
#rows = #of processes in this group
#cols = size

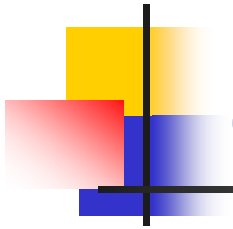
Notes: All processes get a new rank in the new group, and we need to convert the group to a communicator.

```
 1  int numCols;                /* initialized elsewhere */
 2
 3  void broadcast_example()
 4  {
 5    int **ranks;              /* the ranks that belong to each group */
 6    int myRank;
 7    int rowNumber;            /* row number of this process */
 8    int random;               /* value that we would like to broadcast */
 9    rowNumber=myRank/numCols;
10    MPI_Group globalGroup, newGroup;
11    MPI_Comm rowComm[numCols];
12
13    /* initialize ranks[][] array */
14    ranks[0]={0,1,2,3};  /* not legal C */
15    ranks[1]={4,5,6,7};
16    ranks[2]={8,9,10,11};
17    ranks[3]={12,13,14,15};
18
19    /* Extract the original group handle */
20    MPI_Comm_group(MPI_COMM_WORLD, &globalGroup);
21
22    /* Define the new group */
23    MPI_Group_incl(globalGroup, P/numCols, ranks[rowNumber], &newGroup);
24
25    /* Create new communicator */
26    MPI_Comm_create(MPI_COMM_WORLD, newGroup, &newComm);
27
28    random=rand();
29
30    /* Broadcast 'random' across rows */
31    MPI_Bcast(&random, 1, MPI_, rowNumber*numCols, newComm);
32  }
```



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.



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?

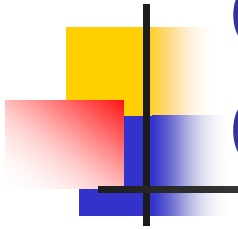
Rank-Coordinates Conversion

- 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)
```

Testing Completion

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

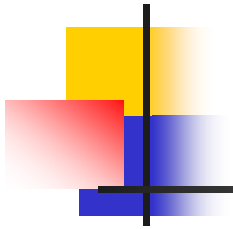
```
int MPI_Test(MPI_Request *request,
             int *flag, MPI_Status *status)
```

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

Previous Example: Safe Program

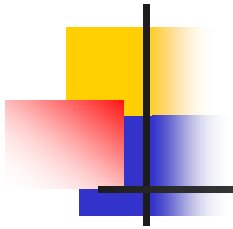
```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Isend(a, 10, MPI_INT, 1, 1, …);
    MPI_Isend(b, 10, MPI_INT, 1, 2, …);
}
else if (myrank == 1) {
    MPI_Irecv(b, 10, MPI_INT, 0, 2, …);
    MPI_Irecv(a, 10, MPI_INT, 0, 1, …);
}
```

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



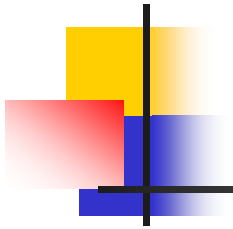
Collective Operation – Later

- 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.



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.

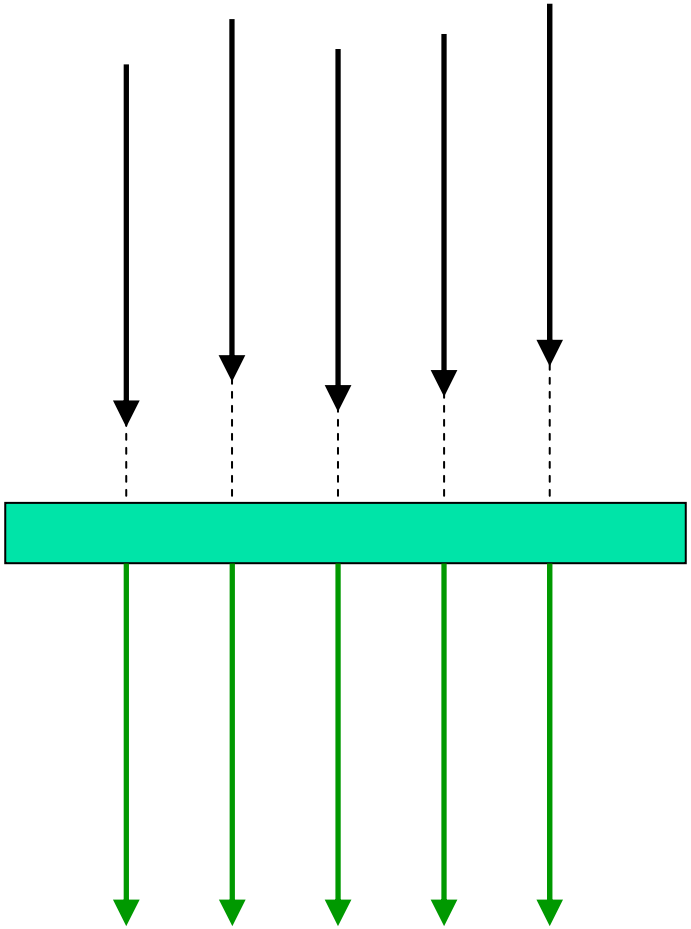


Barrier

- 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)
```

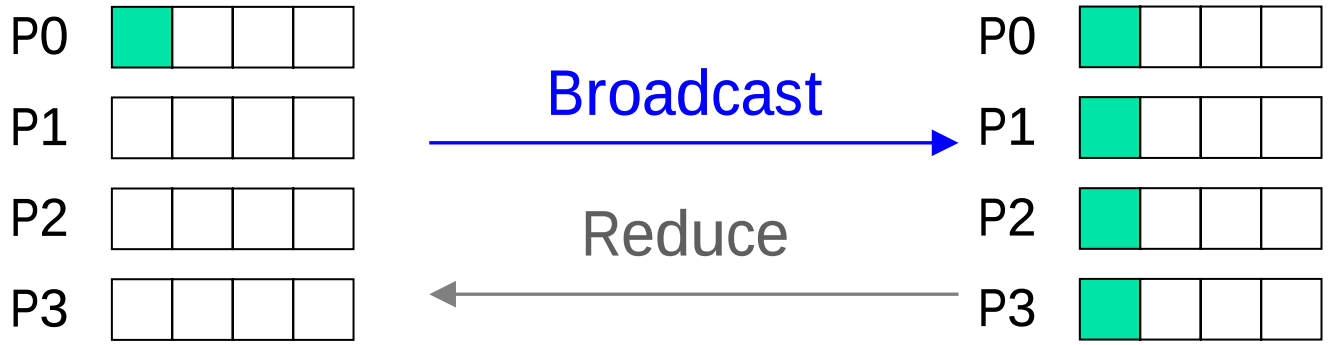
Barrier



One-to-All Broadcast

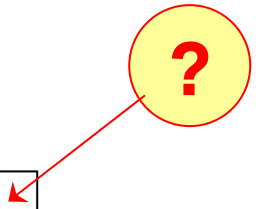
- All the processes must call this function, even the receivers.

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



All-to-One Reduction

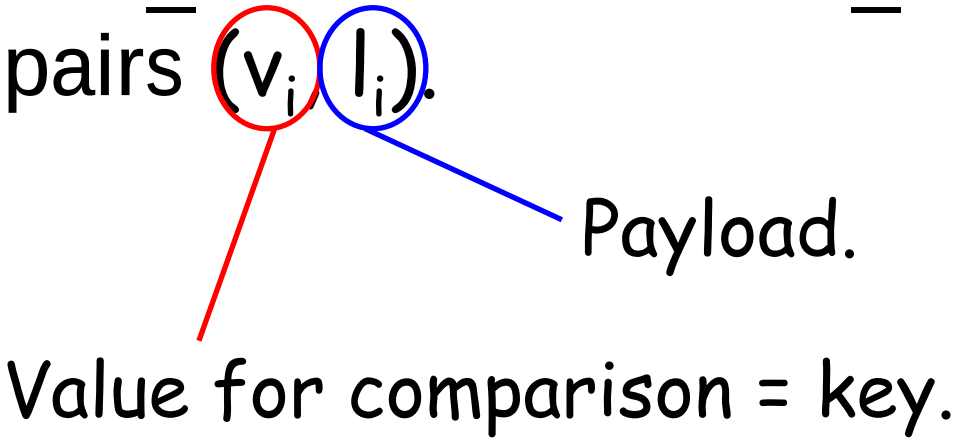
- Combine elements in sendbuf (of each process in the group) using the operation op and return in recvbuf of target.
- Predefined operations: min, max, sum,... associative.




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

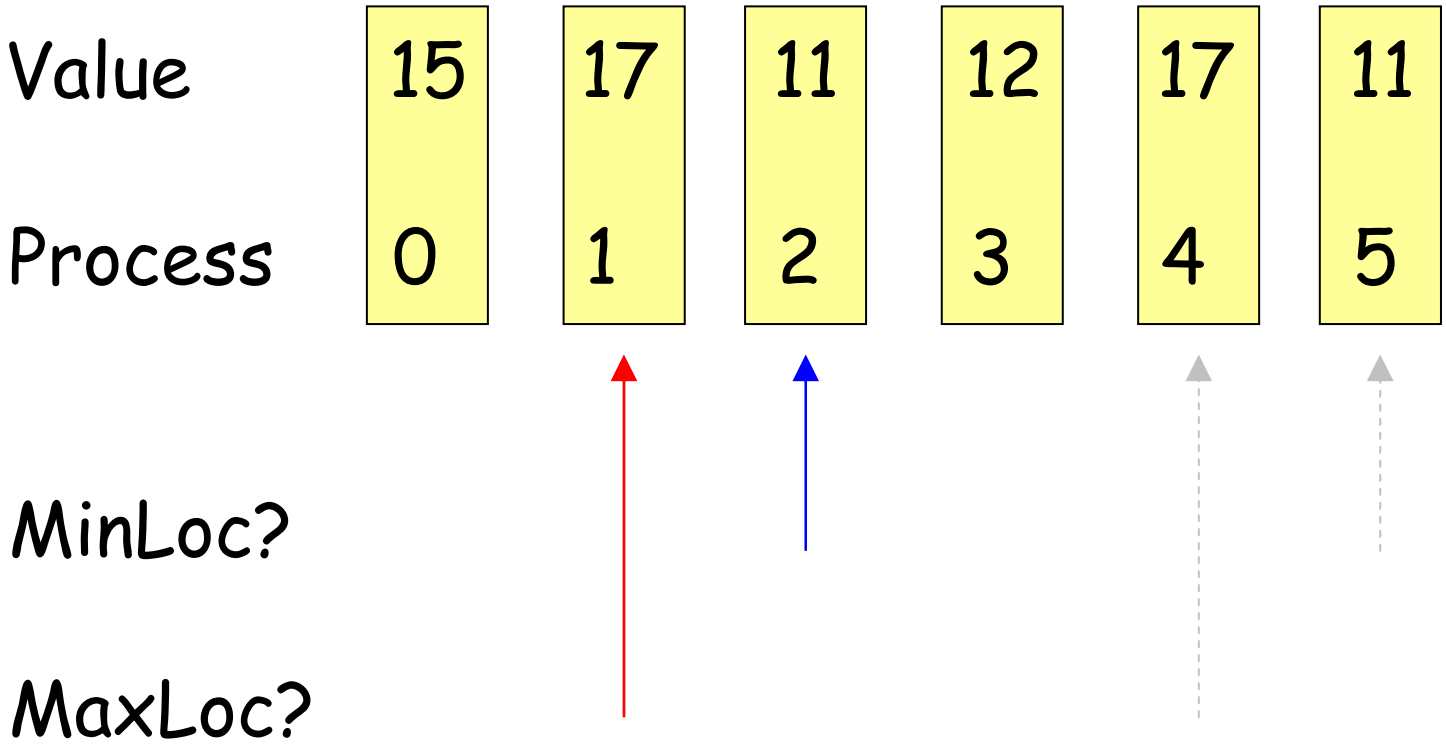
Special Operations

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



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

Example



All-Reduce

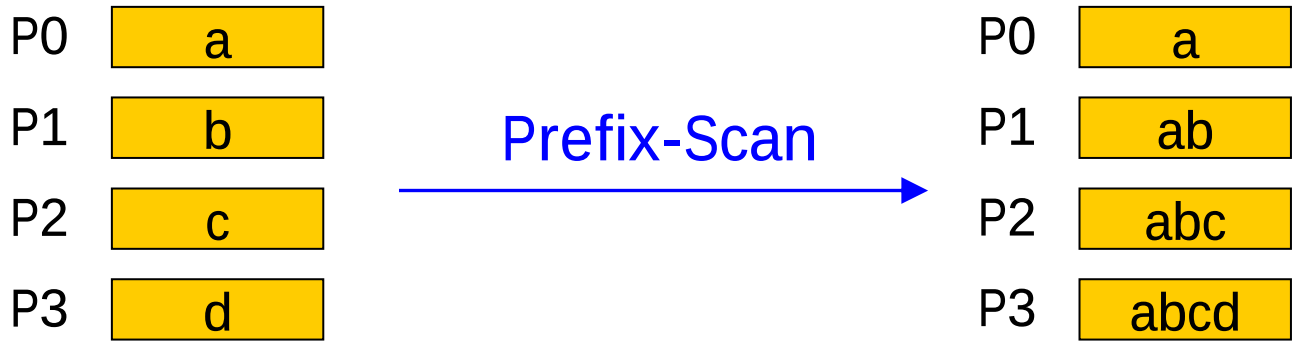
- No target argument since all processes receive the result.

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

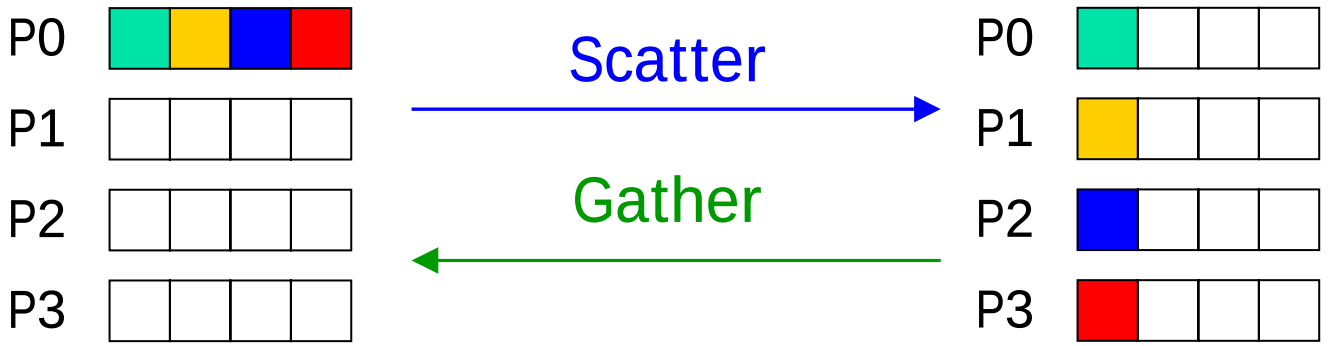
Prefix-Operations

- 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

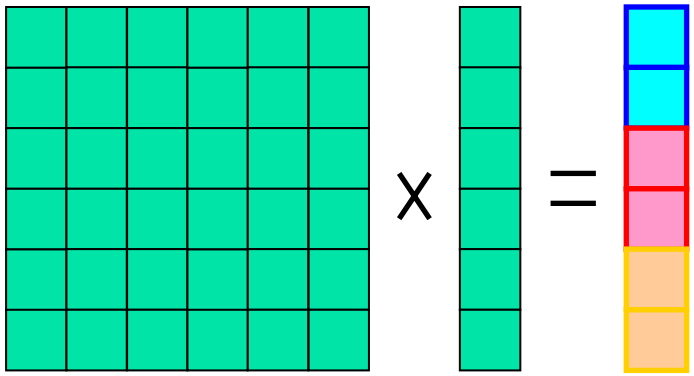


All-Gather

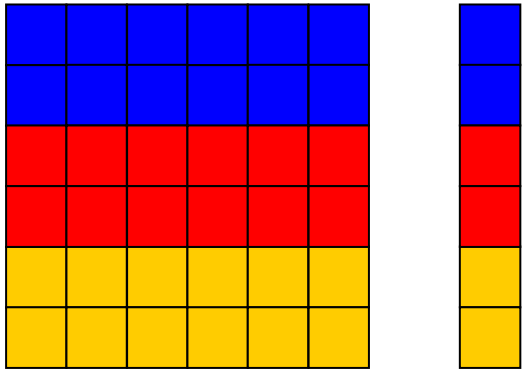
- Variant of gather.

All-to-All Personalized

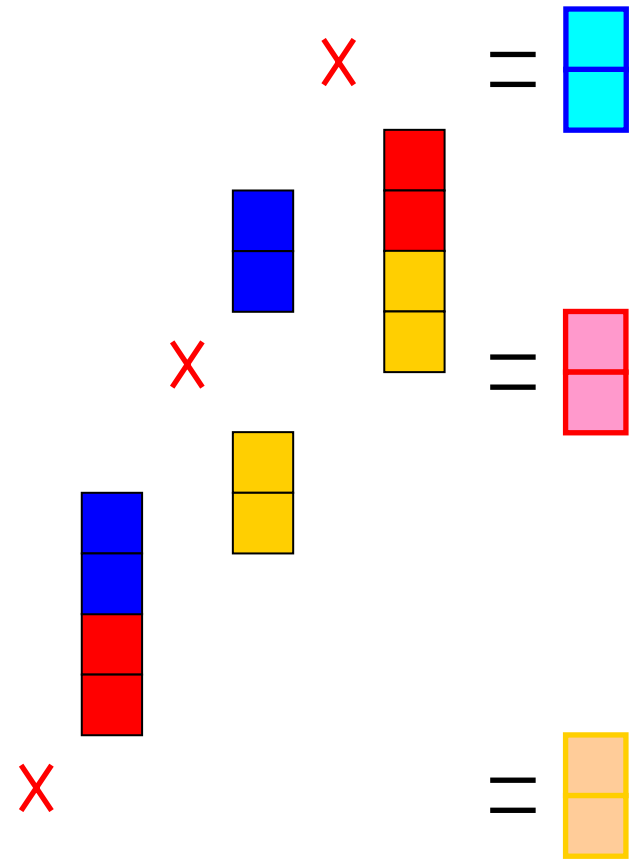
Example Matrix*Vector

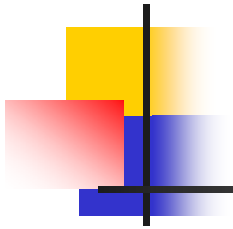


Partition on rows.



Allgather (All-to-all broadcast)



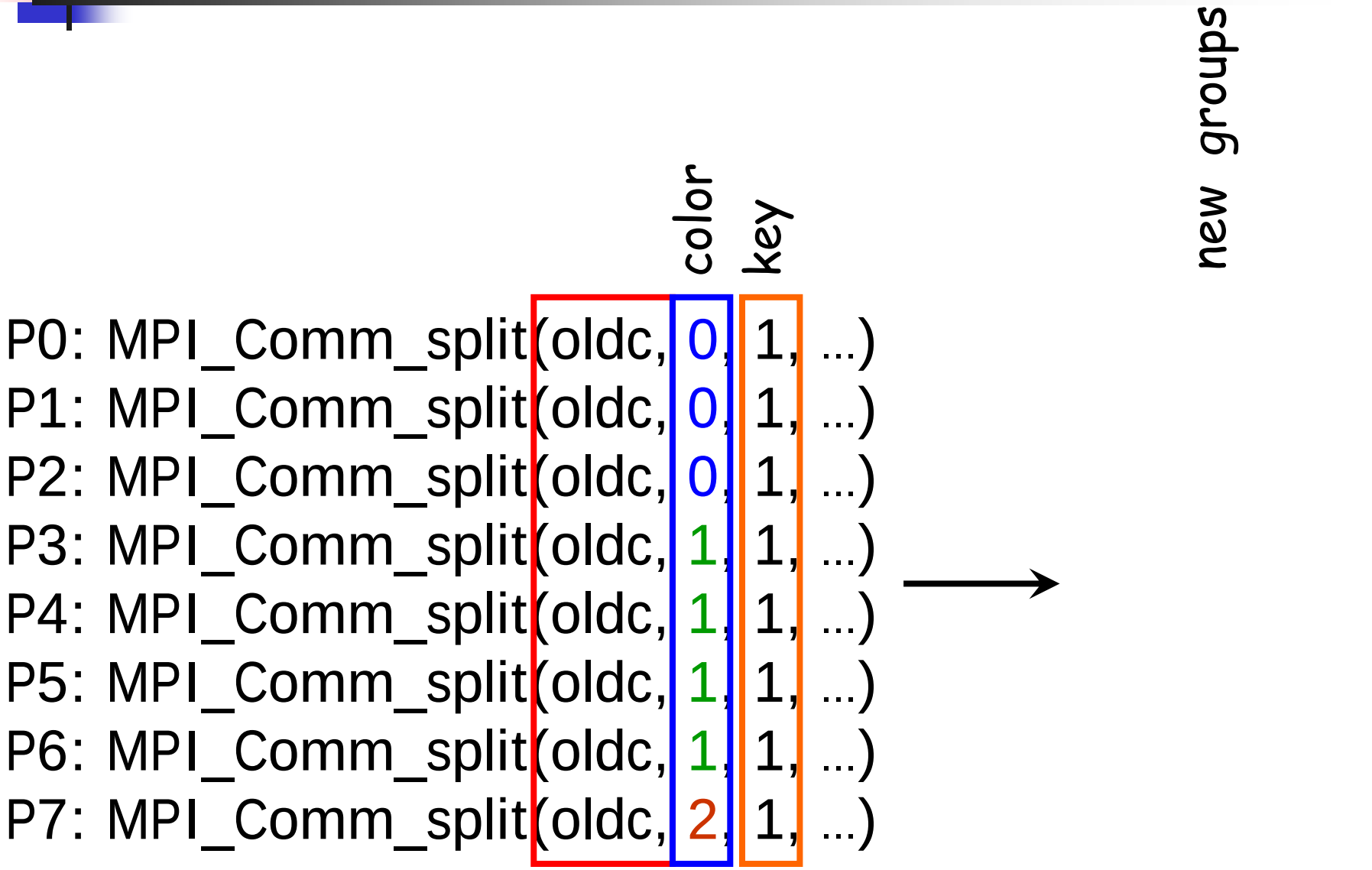


Groups and Communicators

- 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)
```

Split Example



Splitting Cartesian Topologies

- Split Cartesian topology into lower dimensional grids.

Original group.

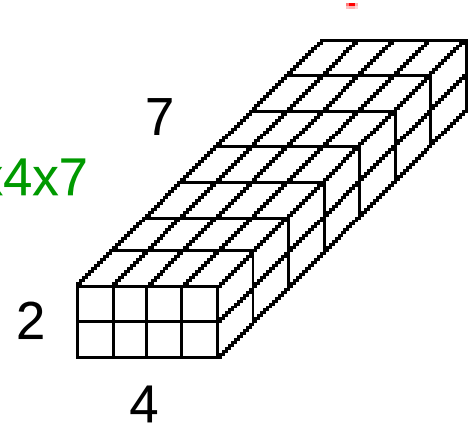
```
int MPI_Cart_sub(MPI_Comm comm_cart,
        int *keep_dims, MPI_Comm *comm_subcart)
```

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

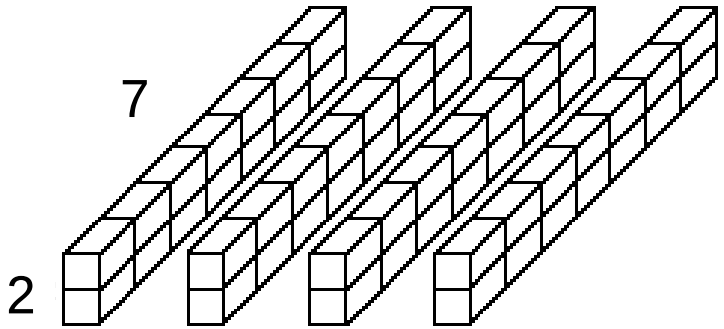
New group.

Example

original 2x4x7

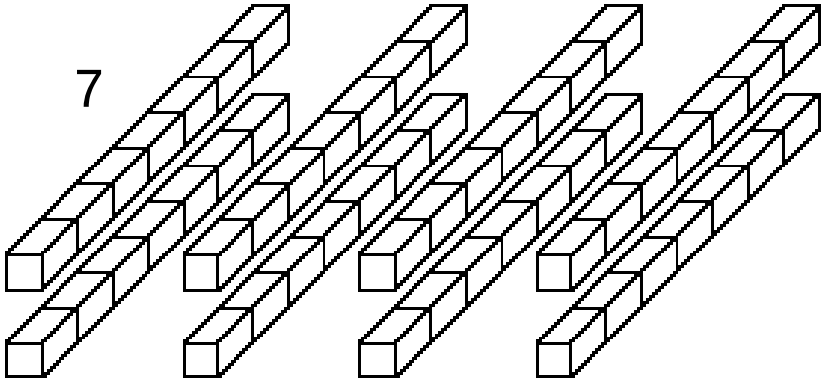


(1,0,1) -> 4* (2x7)



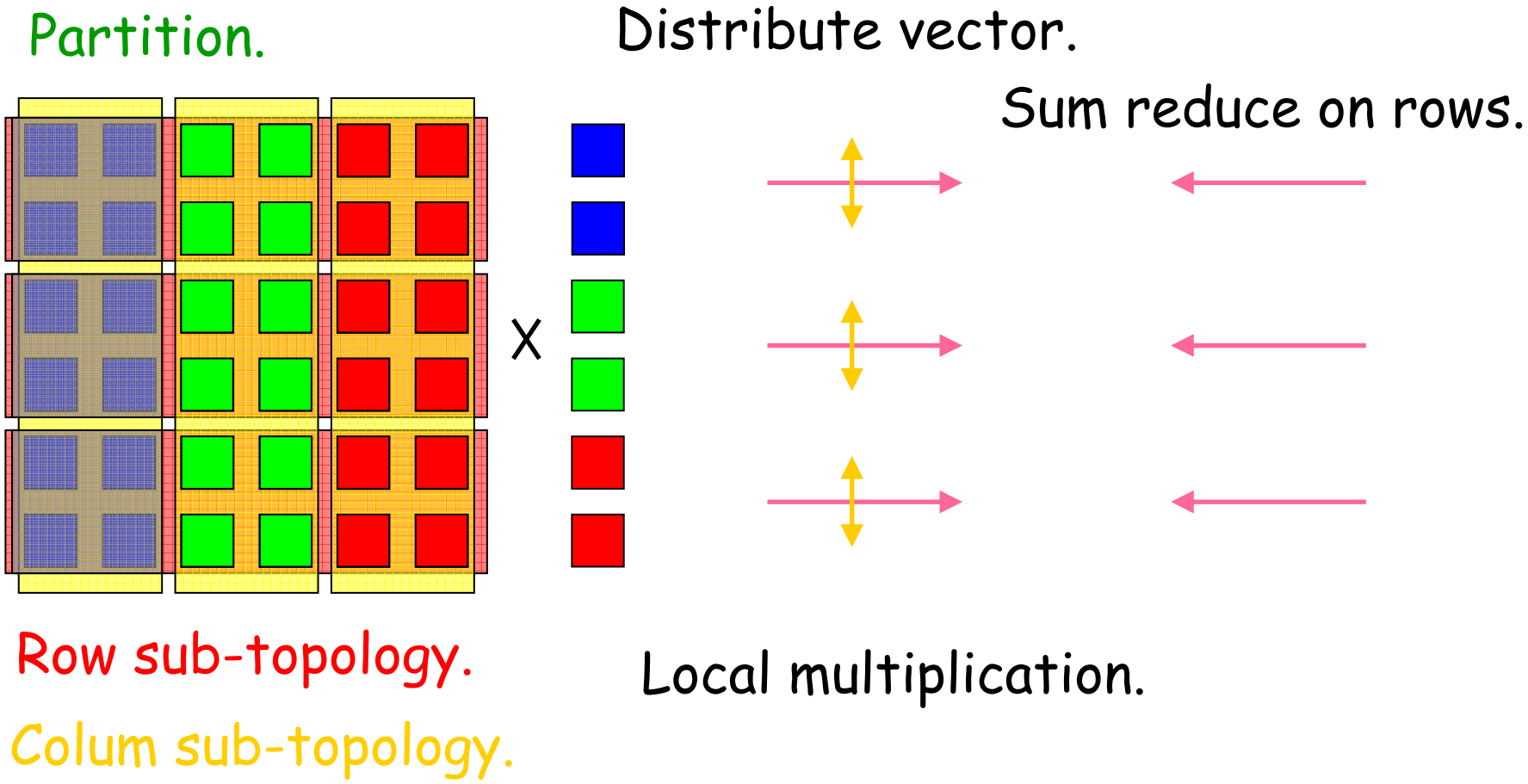
(a)

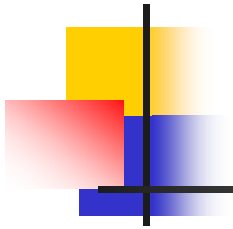
(0,0,1) -> 2*4* (7)



(b)

Example Matrix*Vector





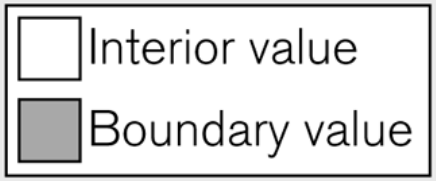
Performance Evaluation

- Elapsed time.

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

Example: Successive Over-Relaxation

1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0



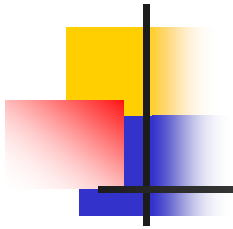
Spot the problem

```
101  do
102  { /*
103    * Send data to four neighbors
104    */
105    if(row !=Top)                                    /* Send North */
106    {
107      MPI_Send(&val[1][1], Width-2, MPI_FLOAT,
108               NorthPE(myID), tag, MPI_COMM_WORLD);
109    }
110
111    if(col !=Right)                                  /* Send East */
112    {
113      for(i=1; i<Height-1; i++)
114      {
115        buffer[i-1]=val[i][Width-2];
116      }
117      MPI_Send(buffer, Height-2, MPI_FLOAT,
118               EastPE(myID), tag, MPI_COMM_WORLD);
119    }
120
121    if(row !=Bottom)                                 /* Send South */
122    {
123      MPI_Send(&val[Height-2][1], Width-2, MPI_FLOAT,
124               SouthPE(myID), tag, MPI_COMM_WORLD);
125    }
126
127    if(col !=Left)                                   /* Send West */
128    {
```

```
129       for(i=1; i<Height-1; i++)
130       {
131         buffer[i-1]=val[i][1];
132       }
133       MPI_Send(buffer, Height-2, MPI_FLOAT,
134                WestPE(myID), tag, MPI_COMM_WORLD);
135     }
136
137     /*
138      * Receive messages
139      */
140     if(row !=Top)                          /* Receive from North */
141     {
142       MPI_Recv(&val[0][1], Width-2, MPI_FLOAT,
143                NorthPE(myID), tag, MPI_COMM_WORLD, &status);
144     }
145
146     if(col !=Right)                        /* Receive from East */
147     {
148       MPI_Recv(&buffer, Height-2, MPI_FLOAT,
149                EastPE(myID), tag, MPI_COMM_WORLD, &status);
150       for(i=1; i<Height-1; i++)
151       {
152         val[i][Width-1]=buffer[i-1];
153       }
154     }
155
156     if(row !=Bottom)                       /* Receive from South */
```

```
155
156     if(row !=Bottom)                        /* Receive from South */
157     {
158       MPI_Recv(&val[0][Height-1], Width-2, MPI_FLOAT,
159                SouthPE(myID), tag, MPI_COMM_WORLD, &status);
160     }
161
162     if(col !=Left)                          /* Receive from West */
163     {
164       MPI_Recv(&buffer, Height-2, MPI_FLOAT,
165                WestPE(myID), tag, MPI_COMM_WORLD, &status);
166       for(i=1; i<Height-1; i++)
167       {
168         val[i][0]=buffer[i-1];
169       }
170     }
171
172     delta=0.0; /* Calculate average, delta for all points */
173     for(i=1; i<Height-1; i++)
174     {
175       for(j=1; j<Width-1; j++)
176       {
```

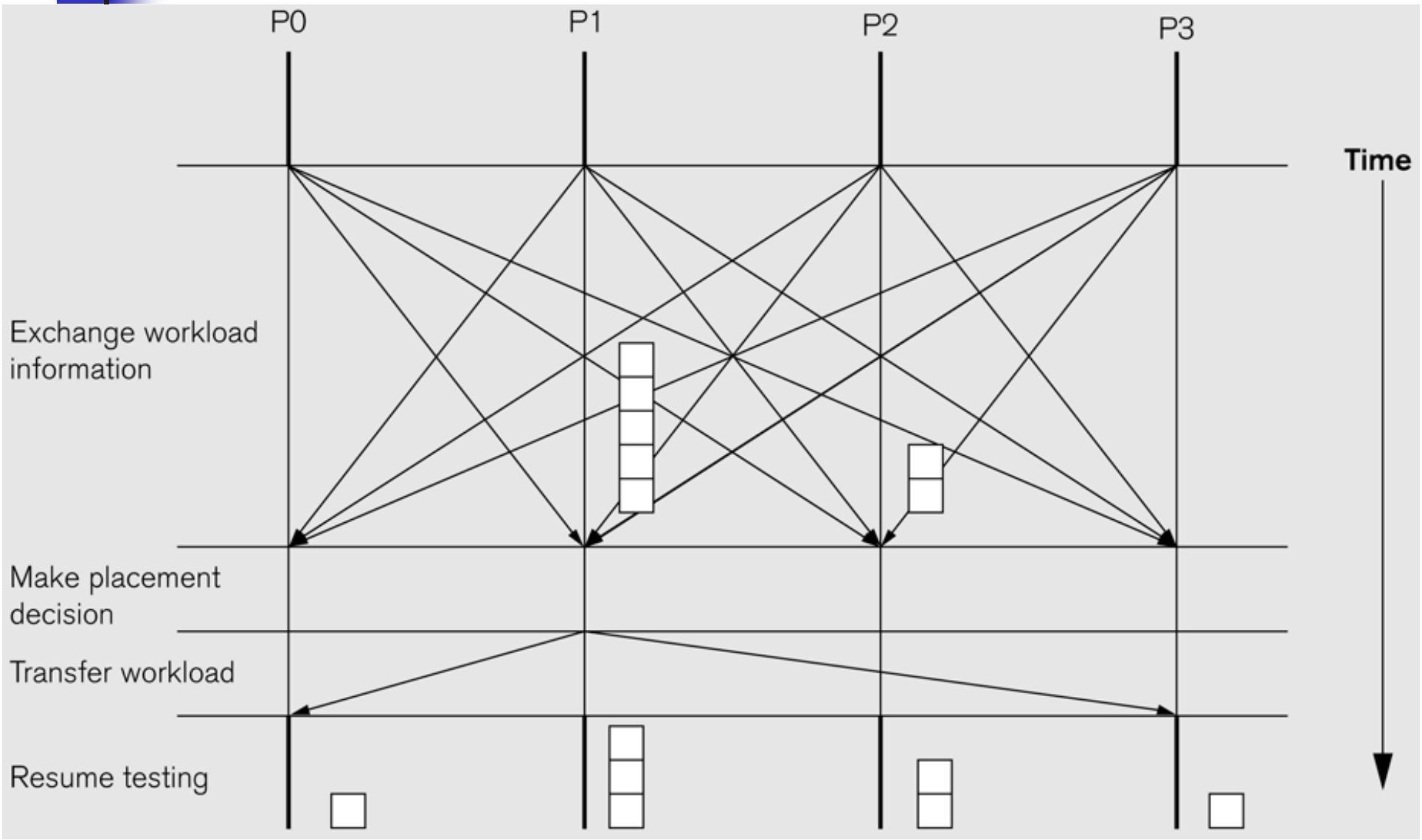
```
177           average=(val[i-1][j]+val[i][j+1]+
178                    val[i+1][j]+val[i][j-1])/4;
179           delta=Max(delta, Abs(average-val[i][j]));
180           new[i][j]=average;
181         }
182       }
183
184       /* Find maximum diff */
185       MPI_Reduce(&delta, &globalDelta, 1, MPI_FLOAT, MPI_MIN,
186                  RootProcess, MPI_COMM_WORLD);
187       Swap(val, new);
188     } while(globalDelta < THRESHOLD);
```



Performance Issues

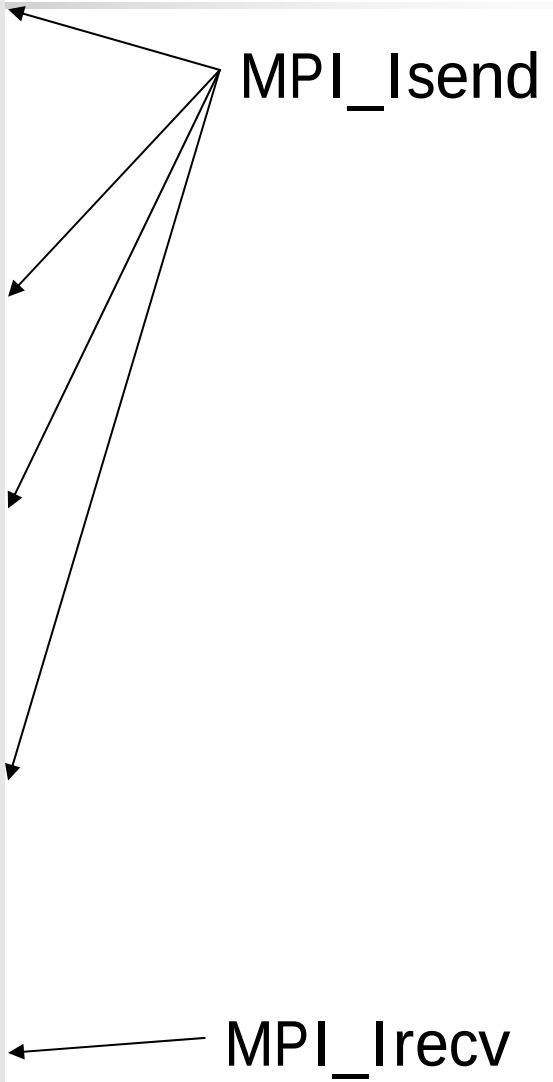
- Large overhead for every message sent.
  - $\rightarrow$ reduce the number of messages.
    Minimize interactions.
    Combine small messages into larger ones.
- MPI programs are static in nature.
  - Large overhead in communication + fixed number of processes.
  - Dynamic work distribution possible but expensive.
- Use non-blocking communication.

Dynamic Work Redistribution



SOR with non-blocking communication

```
101  do
102  {  /*
103     * Send data to four neighbors
104     */
105    if(row!=Top)                    /* Send North */
106    {
107      MPI_ISend(&val[1][1], Width-2, MPI_FLOAT,
108                NorthPE(myID), tag, MPI_COMM_WORLD, &requests[0]);
109    }
110
111    if(col!=Right)                  /* Send East */
112    {
113      for(i=1; i<Height-1; i++)
114      {
115        buffer[i-1]=val[i][Width-2];
116      }
117      MPI_ISend(buffer, Height-2, MPI_FLOAT,
118                EastPE(myID), tag, MPI_COMM_WORLD, &requests[1]);
119    }
120
121    if(row!=Bottom)                 /* Send South */
122    {
123      MPI_ISend(&val[Height-2][1], Width-2, MPI_FLOAT,
124                SouthPE(myID), tag, MPI_COMM_WORLD, &requests[2]);
125    }
126
127    if(col!=Left)                   /* Send West */
128    {
129      for(i=1; i<Height-1; i++)
130      {
131        buffer[i-1]=val[i][1];
132      }
133      MPI_ISend(buffer, Height-2, MPI_FLOAT,
134                WestPE(myID), tag, MPI_COMM_WORLD, &requests[3]);
135    }
136
137    /*
138     * Receive messages
139     */
140    if(row!=Top)                    /* Receive from North */
141    {
142      MPI_IRecv(&val[0][1], Width-2, MPI_FLOAT,
143                NorthPE(myID), tag, MPI_COMM_WORLD, &requests[4]);
144    }
145
146    if(col!=Right)                  /* Receive from East */
147    {
```



```
148       MPI_IRecv(&buffer, Height-2, MPI_FLOAT,
149                 EastPE(myID), tag, MPI_COMM_WORLD, &requests[5]);
150       for(i=1; i<Height-1; i++)
151       {
152         val[i][Width-1]=buffer[i-1];
153       }
154     }
155
156     if(row!=Bottom)                 /* Receive from South */
157     {
158       MPI_IRecv(&val[0][Height-1], Width-2, MPI_FLOAT,
159                 SouthPE(myID), tag, MPI_COMM_WORLD, &requests[6]);
160     }
161
162     if(col!=Left)                   /* Receive from West */
163     {
164       MPI_IRecv(&buffer, Height-2, MPI_FLOAT,
165                 WestPE(myID), tag, MPI_COMM_WORLD, &requests[7]);
166       for(i=1; i<Height-1; i++)
167       {
168         val[i][0]=buffer[i-1];
169       }
170     }
171
172     delta=0.0; /* Calculate average, delta for all points */
173     for(i=2; i<Height-2; i++)
174     {
175       for(j=2; j<Width-2; j++)
176       {
177         average=(val[i-1][j]+val[i][j+1]+
178                  val[i+1][j]+val[i][j-1])/4;
179         delta=Max(delta, Abs(average - val[i][j]));
180         new[i][j]=average;
181       }
182     }
183     MPI_Waitall(8, requests, status);
184
185     /* update top and bottom edges, including corners */
186     for(j=1; j<Width-1; j++)
187     {
188       i=1;
189       average=(val[i-1][j]+val[i][j+1]+
190                val[i+1][j]+val[i][j-1])/4;
191       delta=Max(delta, Abs(average-val[i][j]));
192       new[i][j]=average;
193
194       i=Height-2;
195       average=(val[i-1][j]+val[i][j+1]+
196                val[i+1][j]+val[i][j-1])/4;
```

MPI_Irecv

Compute inner values.

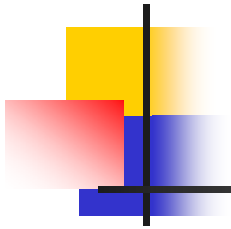
MPI_Waitall

Compute top + bottom.

```
197       delta=Max(delta, Abs(average-val[i][j]));
198       new[i][j]=average;
199     }
200
201     /* update left and right edges, excluding corners */
202     for(i=2; i<Height-2; i++)
203     {
204       j=1;
205       average=(val[i-1][j]+val[i][j+1]+
206                  val[i+1][j]+val[i][j-1])/4;
207       delta=Max(delta, Abs(average - val[i][j]));
208       new[i][j]=average;
209
210       j=Width-2;
211       average=(val[i-1][j]+val[i][j+1]+
212                  val[i+1][j]+val[i][j-1])/4;
213       delta=Max(delta, Abs(average-val[i][j]));
214       new[i][j]=average;
215     }
216     /* Find maximum diff */
217     MPI_Reduce(&delta, &globalDelta, 1, MPI_FLOAT, MPI_MIN,
218                RootProcess, MPI_COMM_WORLD);
219     Swap(val, new);
220   } while(globalDelta < THRESHOLD);
```

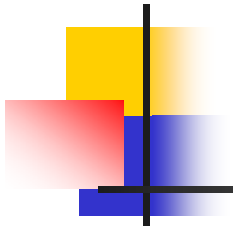
Compute left+right.

MPI_Reduce



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

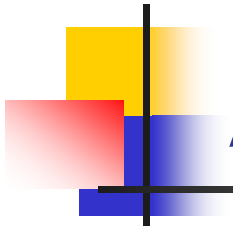


MPI In Practice

- 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



Alternatives to MPI

- Partitioned global address space languages.
  - Abstract communication.
  - Use local & non-local memories.
  - Conceptually close to Peril-L.
  - Examples:
    - Co-Array Fortran
    - Unified Parallel C
    - Titanium
- Older distributed shared memory systems
  - Linda: distributed tuple space.