Numerical Simulation and Scientific Computing II

Lecture 2:
Distributed Parallel Computing II

Josef Weinbub, Paul Manstetten,
Heinz Pettermann, Asur Vijaya Kumar, Pavan
Kumar, Jesús Carrete Montana, Francesco
Zonta, Kevin Sturm

Institute for Microelectronics
TU Wien

nssc@iue.tuwien.ac.at

SS 2023

Quiz

Q1: How is it ensured that a specific message is received by a specific process?

→ ***tag*** **and** ***dest*** **parameter**

Q2: What is the first and last routine to be called in a MPI program?

→ ***MPI_Init (…)*** **and** ***MPI_Finalize()***

Q3: Is "MPI_Init" executed by one, several or all MPI processes?

→ **all**

Q4: Name typical reduction operations?

→ **sum, min, max, etc.**

Q5: How can a point-to-point communication be made non-blocking? What potential advantage is there?

→ **MPI_I…, potentially allows to overlap communication with computation**

Outline

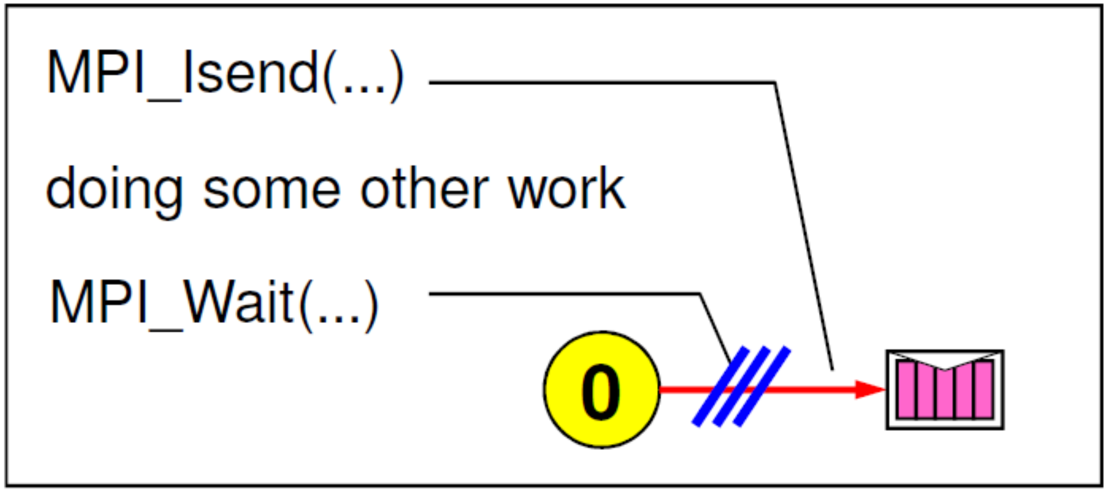
- Distributed Parallel Computing II
  - **Non-Blocking Communication**
  - Collective Communication
  - Derived Datatypes
  - Virtual Topologies
  - Hybrid MPI+OpenMP
- Quiz

Non-Blocking Communications

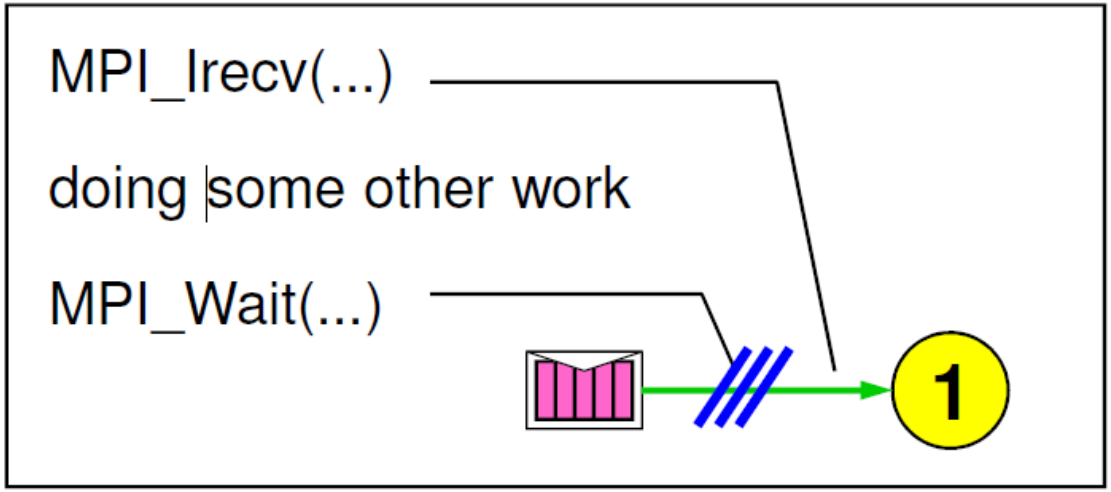
- **Separate communication into three phases:**
  - **Initiate nonblocking communication**
    - **returns Immediately**
    - **routine name starting with MPI_I…**
  - **Do some work (perhaps involving other communications?)**
    - **If cluster hardware doesn't have dedicated MPI communication logic, don't expect efficient overlapping of communication with computation: Communication requires significant computational resources!**
  - **Wait for non-blocking communication to complete**

Non-Blocking Examples

- **Non-blocking <u>send</u>**



- **Non-blocking <u>receive</u>**

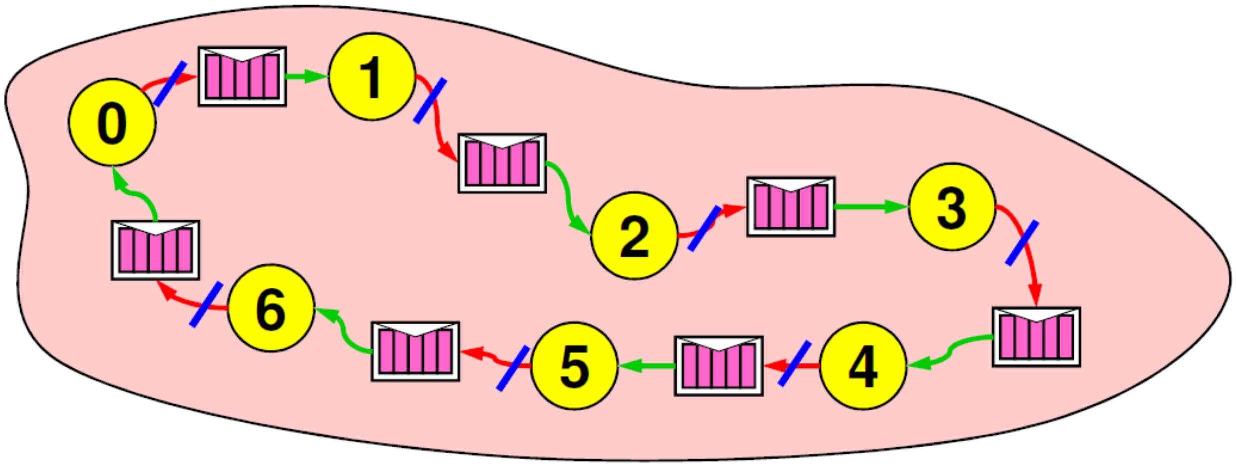


Non-Blocking Send

- **Initiate non-blocking send**
  - **Ring example: Initiate non-blocking send to the right neighbor**

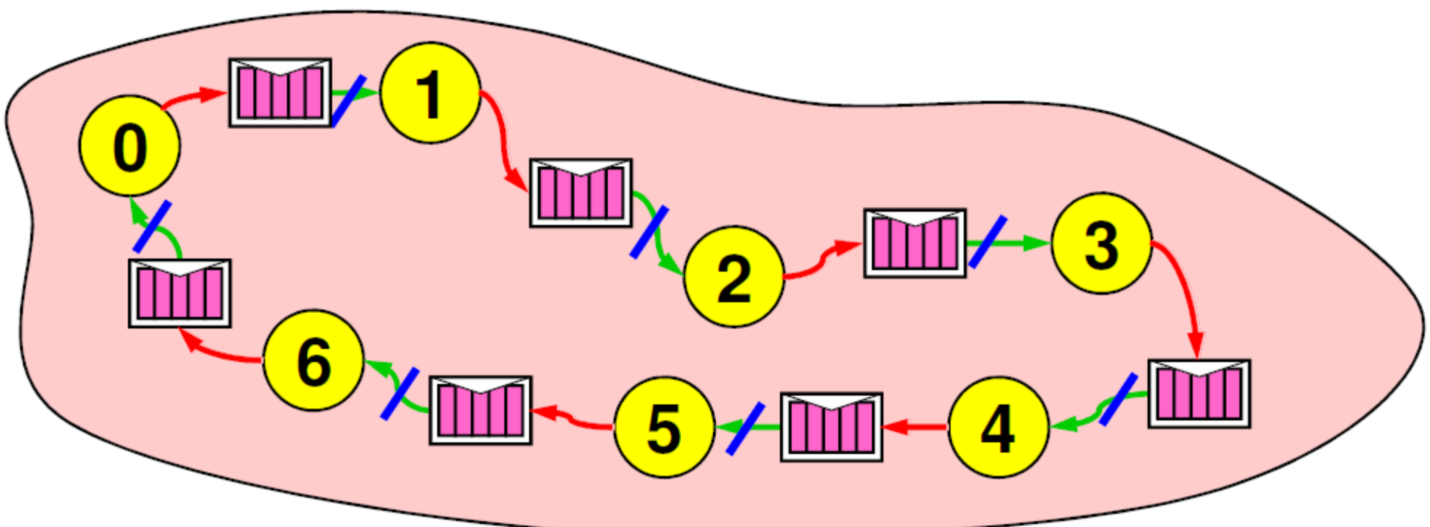
- **Do some work**
  - **Ring example: Receiving the message from left neighbor**

- **Message transfer can be completed**
- **Wait for non-blocking send to complete**



Non-Blocking Receive

- **Initiate non-blocking receive**
  - **Ring example: Initiate non-blocking receive from left neighbor**
- **Do some work**
  - **Ring example: Sending the message to the right neighbor**
- **Message transfer can be completed**
- **Wait for non-blocking receive to complete**



Request Handles

- **Request handles**
  - **Are used for non-blocking communication**
  - **<u>Must</u> be stored in local variables: MPI_Request**
    - **is generated by a non-blocking communication routine**
    - **is used (and freed) in the MPI_WAIT routine**

Non-Blocking Synchronous Send

- **buf must not be modified between Issend and Wait**
- **"Issend + Wait directly after" is equivalent to blocking call (Ssend)**
- **status is not used in Issend, but in Wait (with send: nothing returned)**

```
MPI_Issend( buf, count, datatype, dest, tag, comm, [OUT] &request_handle);
      ⬇
MPI_Wait( [INOUT] &request_handle, &status);
```

Nonblocking Receive

- **<u>buf</u> must not be used between <u>Irecv</u> and <u>Wait</u>**

```
MPI_Irecv ( buf, count, datatype, source, tag, comm, [OUT] &request_handle);
MPI_Wait( [INOUT] &request_handle, &status);
```

Blocking and Non-Blocking

- Send and receive can be blocking or non-blocking.
- A blocking send can be used with a non-blocking receive, and vice-versa.
- Non-blocking sends can use any mode
  - standard – MPI_ISEND
  - synchronous – MPI_ISSEND
  - buffered – MPI_IBSEND
  - ready – MPI_IRSEND
- Synchronous mode affects completion, i.e. MPI_Wait / MPI_Test, not initiation, i.e., MPI_I...
- The non-blocking operation immediately followed by a matching wait is equivalent to the blocking operation.

Completion

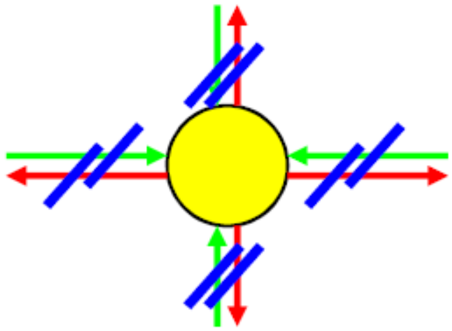
```
MPI_Wait( &request_handle, &status);
MPI_Test( &request_handle, &flag, &status);
```

- **You need to**
  - **WAIT or**
  - **Loop with TEST until request is completed, i.e., flag == 1**

Multiple Non-Blocking Communications

You have several request handles:

- Wait or test for completion of one message
  - MPI_Waitany / MPI_Testany
- Wait or test for completion of all messages
  - MPI_Waitall / MPI_Testall *)
- Wait or test for completion of as many messages as possible (i.e., at least one)
  - MPI_Waitsome / MPI_Testsome *)



*) Each status contains an additional error field.

This field is only used if MPI_ERR_IN_STATUS is returned (also valid for send operations).

Send-Receive in One Routine

- **MPI_Sendrecv & MPI_Sendrecv_replace**
  - **Combines the triple "MPI_Irecv + Send + Wait" into one routine**

Performance Options

Which is the fastest neighbor communication?

- **MPI_Irecv + MPI_Send**
- **MPI_Irecv + MPI_Isend**
- **MPI_Isend + MPI_Recv**
- **MPI_Isend + MPI_Irecv**
- **MPI_Sendrecv**
- **MPI_Neighbor_alltoall**

No answer by the MPI standard, because:

MPI targets portable and efficient message-passing programming but efficiency of MPI application-programming is not portable!

Example Non-Blocking Communication

- **Integration with MPI non-blocking communications**
  - **Example source:** http://www.bu.edu/tech/support/research/training-consulting/online-tutorials/mpi/example1-2/example1_3/
  - **Background on numerical integration:** http://www.bu.edu/tech/support/research/training-consulting/online-tutorials/mpi/example1-2/
- **Until a matching receive has signaled that it is ready to receive, a blocking send will continue to wait.**

```
void other_work(int myid) {
  printf("more work on process %dn", myid);
}
float integral(float ai, float h, int n){
  int j;
  float aij, integ;
  integ = 0.0;              /* initialize */
  for (j=0;j<j++) {         /* sum integrals */
    aij = ai + (j+0.5)*h;      /* mid-point */
    integ += cos(aij)*h;
  }
  return integ;
}
```

Support Functions

Example Non-Blocking Communication

Common Part

```
int n, p, myid, tag, master, proc, ierr;
float h, integral_sum, a, b, ai, pi, my_int;
MPI_Comm comm;
MPI_Request request;
MPI_Status status;
comm = MPI_COMM_WORLD;
ierr = MPI_Init(&argc,&argv);          /* starts MPI */
MPI_Comm_rank(comm, &myid);            /* get current process id */
MPI_Comm_size(comm, &p);               /* get number of processes */
master = 0;
pi = acos(-1.0);  /* = 3.14159... */
a = 0.;           * lower limit of integration */
b = pi*1./2.;     /* upper limit of integration */
n = 500;          /* number of increment within each process */
tag = 123;        /* set the tag to identify this particular job */
h = (b-a)/n/p;    /* length of increment */
ai = a + myid*n*h;  /* lower limit of integration for partition myid */
my_int = integral(ai, h, n);    /* 0<=myid<=p-1 */
printf("Process %d has the partial result of %fn", myid, my_int);
```

Example Non-Blocking Communication

Master Part

```
if(myid == master) {
  integral_sum = my_int;
  for (proc=1;proc<p;proc++) {
    MPI_Recv(
            &my_int, 1, MPI_FLOAT,    /* triplet of buffer, size, data type */
            MPI_ANY_SOURCE,       /* message source */
            MPI_ANY_TAG,          /* message tag */
            comm, &status);       /* status identifies source, tag */
    integral_sum += my_int;
  }
  printf("The Integral =%fn",integral_sum); /* sum of my_int */
}
```

Example Non-Blocking Communication

Worker Part

```
 else {
   MPI_Isend(      /* non-blocking send */
           &my_int, 1, MPI_FLOAT,      /* triplet of buffer, size, data type */
           master,
           tag,
           comm, &request);      /* send my_int to master */
   other_work(myid);
   MPI_Wait(&request, &status);    /* block until Isend is done */
  }
  MPI_Finalize();                 /* let MPI finish up ... */
}
```

Outline

- Distributed Parallel Computing II
  - Non-Blocking Communication
  - **Collective Communication**
  - Derived Datatypes
  - Virtual Topologies
  - Hybrid MPI+OpenMP
- Quiz

Collective Communication

- **Communications involving a group of processes.**
- **Called by all processes in a communicator.**
- **Examples:**
  - **Barrier synchronization.**
  - **Broadcast, scatter, gather.**
  - **Global sum, global maximum, etc.**
  - **Neighbor communication in a virtual grid**

Characteristics of Collective Communication

- Collective action over a communicator.
- All processes of the communicator must communicate, i.e., must call the collective routine.
- Synchronization may or may not occur, therefore all processes must be able to start the collective routine.
- On a given communicator, the n-th collective call must match on all processes of the communicator.
- In MPI-1.0 – MPI-2.2, all collective operations are blocking. Non-blocking versions since MPI-3.0.
  - Not covered in this lecture
- No tags.
- Receive buffers must have exactly the same size as send buffers.

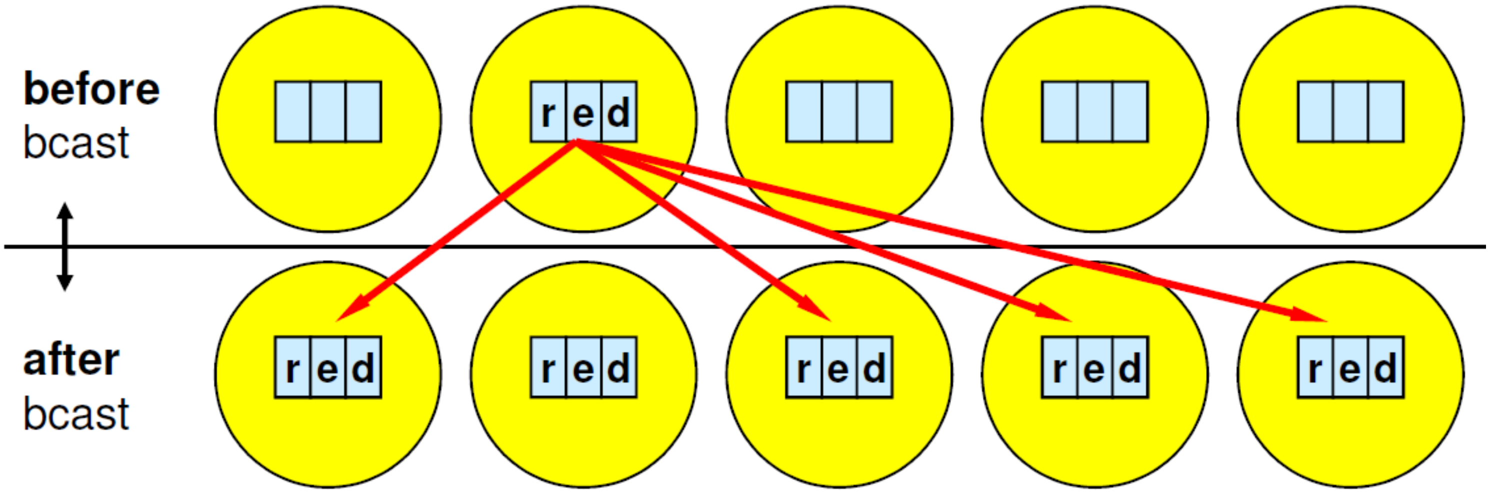
Barrier Synchronization

MPI_Barrier is normally never needed:

- **all synchronization is done automatically by the data communication:**
  a process cannot continue before it has the data that it needs.
- **if used for debugging:**
  please guarantee, that it is removed in production.
- **for profiling: to separate time measurements**

Broadcast

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

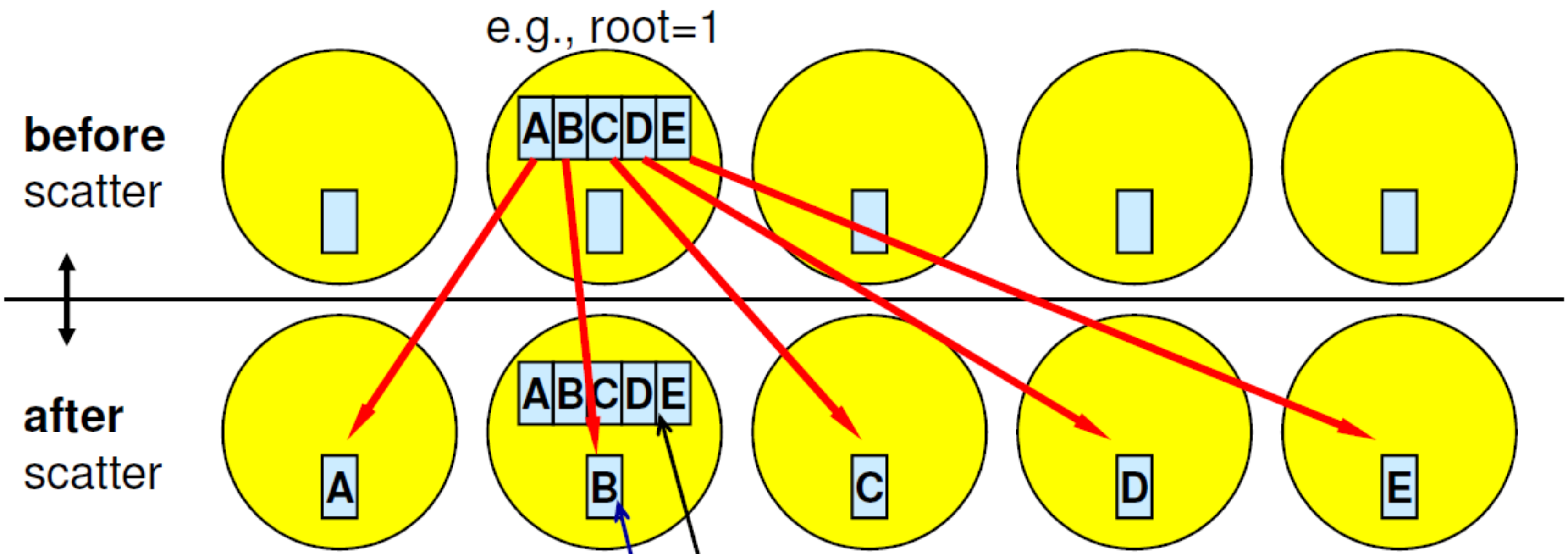


- **E.g. <u>root</u>=1: all processes must use same ”root”**

Example

```
std::vector<char> buf(3);
if(myrank == 1) { buf[0]='r'; buf[1]='e'; buf[2]='d'; }
MPI_Bcast(buf.data(), buf.size(), MPI_CHAR, 1, MPI_COMM_WORLD);
```

Scatter



int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm)

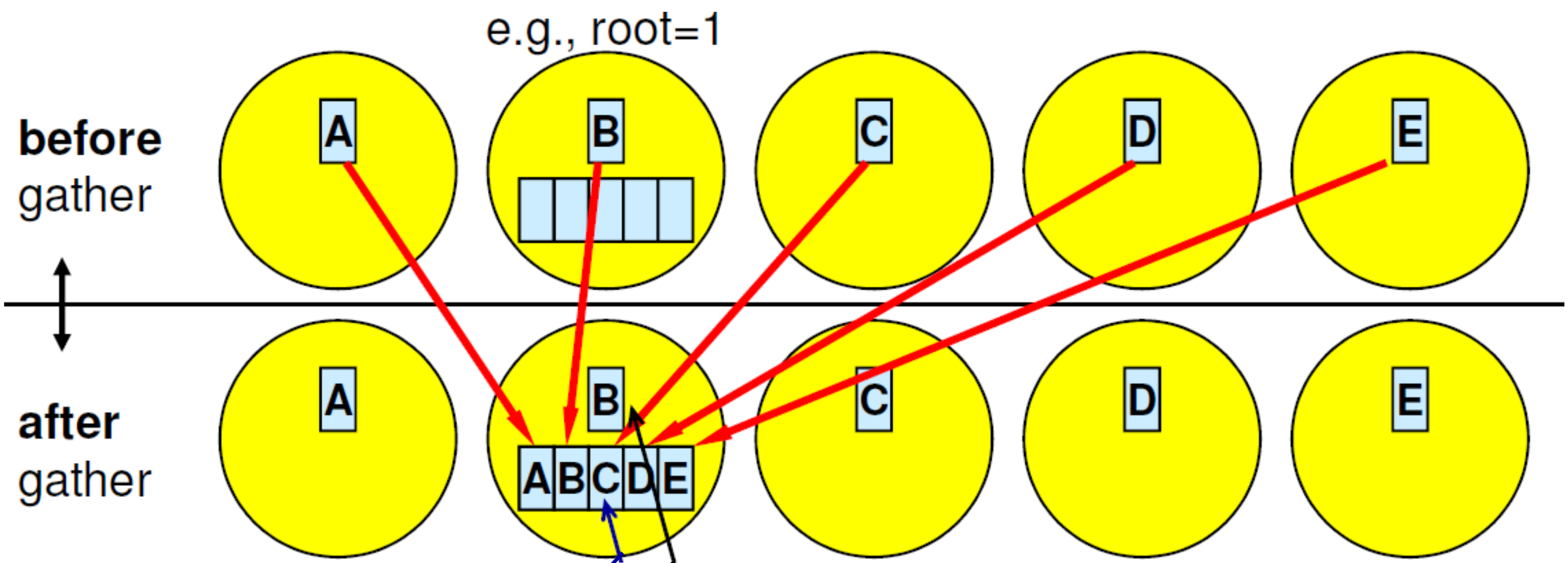
Example

```
std::vector<char> sbuf(5); char rbuf;
if(myrank == 1) sbuf[0]='A'; sbuf[1]='B'; // etc.
MPI_Scatter(sbuf.data(), 1, MPI_CHAR,
            &rbuf, 1, MPI_CHAR,
            1, MPI_COMM_WORLD)
```

If data cannot be equally distributed to all processes, use:
MPI_Scatterv: allows for variable send/recvcounts.

Sum of transmitted data size must match:
sendcount x sendtype =
recvcount x recvtype
In principle: allowed to mix data types!
But typically: counts and types the same.

Gather



int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm)

Example

```
if(myrank==0) char sbuf='A';   // etc...'B', 'C', 'D', 'E'
std::vector<char> rbuf(5);
MPI_Gather(&sbuf, 1, MPI_CHAR,
           rbuf.data(), 1, MPI_CHAR,
           1, MPI_COMM_WORLD)
```

See notes for Scatter, same here.

Global Reduction Operations

- To perform a global reduce operation across
- d0 o d1 o d2 o d3 o … o ds-2 o ds-1
  - di = data in process rank i
    - single variable, or
    - vector
  - o = associative operation
  - Example:
    - global sum or product
    - global maximum or minimum
    - global user-defined operation

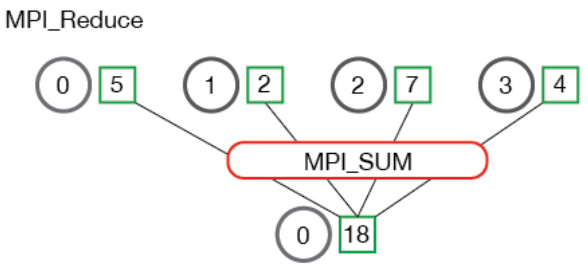
Predefined Reduction Operation Handles

Predefined operation handle	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location of the maximum
MPI_MINLOC	Minimum and location of the minimum

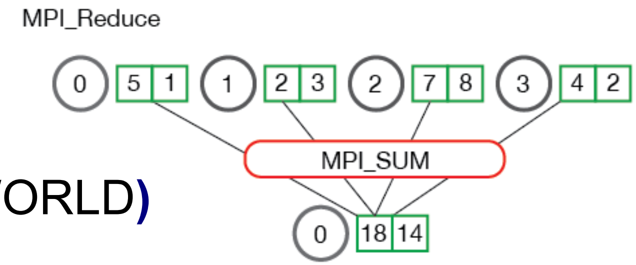
Reduce

```
int MPI_Reduce(void *inbuf, void *result, int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm)
```

```
int inbuf = 5; // 2, 7, 4 for other ranks
int result;
MPI_Reduce(&inbuf, &result, 1, MPI_INT,
           MPI_SUM, 0, MPI_COMM_WORLD)
```




```
std::vector<int> inbuf {5,1}; // {2,3} … for other ranks
std::vector<int> result (2);
MPI_Reduce(inbuf.data(), result.data(), result.size(),
           MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
```



Example Collective Communication

- **Integration with MPI collective communications**
  - **Example source: http://www.bu.edu/tech/support/research/training-consulting/online-tutorials/mpi/example1-2/example1_4/**
  - **Adaption of previous example based on non-blocking communication**

```
float integral(float ai, float h, int n){
  int j;
  float aij, integ;
  integ = 0.0;                /* initialize */
  for (j=0;j<j++) {           /* sum integrals */
    aij = ai + (j+0.5)*h;     /* mid-point */
    integ += cos(aij)*h;
  }
  return integ;
}
```

Support Function

Example Collective Communication

First Part

```
int n, p, myid, tag, proc, ierr, i;
float h, integral_sum, a, b, ai, pi, my_int, buf[50];
int master = 0;  /* processor performing total sum */
MPI_Comm comm;
comm = MPI_COMM_WORLD;
ierr = MPI_Init(&argc,&argv);            /* starts MPI */
MPI_Comm_rank(comm, &myid);              /* get current process id */
MPI_Comm_size(comm, &p);                 /* get number of processes */
pi = acos(-1.0);  /* = 3.14159... */
a = 0.;           /* lower limit of integration */
b = pi*1./2.;     /* upper limit of integration */
n = 500;          /* number of increment within each process */
tag = 123;        /* set the tag to identify this particular job */
h = (b-a)/n/p;    /* length of increment */
ai = a + myid*n*h;  /* lower limit of integration for partition myid */
my_int = integral(ai, h, n);   /* 0<=myid<=p-1 */
printf("Process %d has the partial sum of %fn", myid,my_int);
```

buf-size >= p
(see next slide)

Example Collective Communication

Second Part

```
MPI_Gather(       /* collects my_int from all processes to master */
    &my_int, 1, MPI_FLOAT,       /* send buffer, size, data type */
    &buf[0], 1, MPI_FLOAT,      /* receive buffer, size, data type */
    master, comm);

if(myid == master) {
  integral_sum = 0.0;
  for (i=0; i<p; i++) {
    integral_sum += buf[i];
  }
  printf("The Integral =%fn",integral_sum);
}

MPI_Finalize();
```

buf must fit all ranks!

Is there another collective operation which could be used here?

Yes, MPI_Reduce with MPI_SUM

Outline

- Distributed Parallel Computing II
  - Non-Blocking Communication
  - Collective Communication
  - **Derived Datatypes**
  - Virtual Topologies
  - Hybrid MPI+OpenMP
- Quiz

Derived Datatypes

- Description of the memory layout of the buffer
  - for sending
  - for receiving
- Basic types
- Derived types
  - vectors
  - structs
  - others

Data Layout and the Describing Datatype Handle

```
struct buff_layout
  { int     i_val[3];
    double d_val[5];
  } buffer;
```

Compiler

```
array_of_types[0]=MPI_INT;
array_of_blocklengths[0]=3;
array_of_displacements[0]=
array_of_types[1]=MPI_DOUBLE;
array_of_blocklengths[1]=5;
array_of_displacements[1]=
```

?

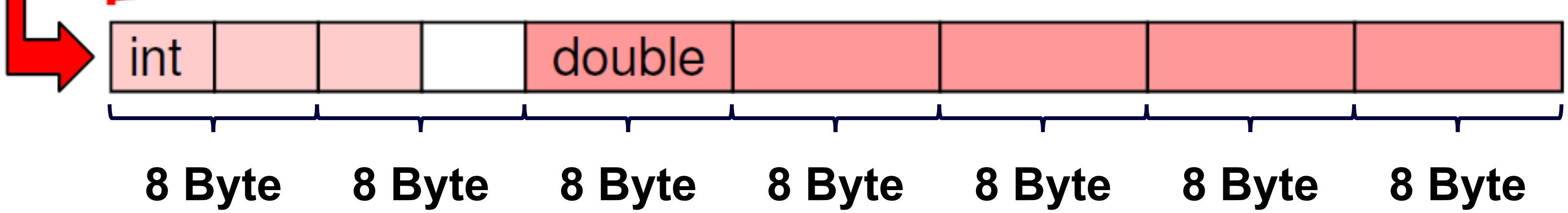
```
MPI_Type_create_struct(2, array_of_blocklengths,
      array_of_displacements, array_of_types,
                         &buff_datatype);

MPI_Type_commit(&buff_datatype);
```

```
MPI_Send(&buffer, 1, buff_datatype, ...)
```

&buffer = the start address of the data

the datatype handle describes the data layout



Compute Displacement

```
array_of_types[0]=MPI_INT;
array_of_blocklengths[0]=3;
array_of_displacements[0]=
array_of_types[1]=MPI_DOUBLE;
array_of_blocklengths[1]=5;
array_of_displacements[1]=
MPI_Type_create_struct(2, array_of_blocklengths,
    array_of_displacements, array_of_types,
                        &buff_datatype);
MPI_Type_commit(&buff_datatype);
```

In principle:

array_of_displacements[i] = address(block_i) – address(block_0);

Use:

int MPI_Get_address(void* location, MPI_Aint *address)

Continuing the example:

```
int address_base, address_0, address_1;
MPI_Get_address(&buffer, &address_base);
MPI_Get_address(&buffer.i_val[0], &address_0);
MPI_Get_address(&buffer.d_val[0], &address_1);
array_of_displacements[0] = address_0 – address_base;
array_of_displacements[1] = address_1 – address_base;
```

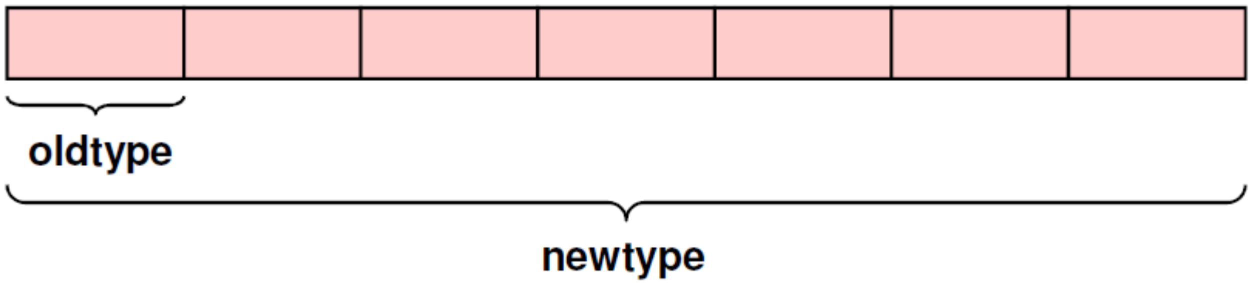
```
struct buff_layout
  { int       i_val[3];
    double d_val[5];
  } buffer;
```

Derived Datatype: Another Example

```
typedef struct {
  char      a;
  int       b;
  double   c;} mystruct;
mystruct mydata;
int baseaddr, addr0, addr1, addr2;
MPI_Get_address ( &mydata,   &baseaddr);
MPI_Get_address ( &mydata.a, &addr0);
MPI_Get_address ( &mydata.b, &addr1);
MPI_Get_address ( &mydata.c, &addr2);
displ[0] = addr0 – baseaddr;
displ[1] = addr1 – baseaddr;
displ[2] = addr2 – baseaddr;
dtype[0] = MPI_CHAR;
blength[0] = 1;
dtype[1] = MPI_INT;
blength[1] = 1;
dtype[2] = MPI_DOUBLE;
blength[2] = 1;
MPI_Type_struct ( 3, blength, displ, dtype, &newtype );
MPI_Type_commit ( &newtype );
```

Contiguous Data

- **The simplest derived datatype**
- **Consists of a number of contiguous items of the same datatype**




```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                        MPI_Datatype *newtype)
```

Example

```
int myvec[4];
MPI_Type_contiguous ( 4, MPI_INT, &mybrandnewdatatype);
MPI_Type_commit ( &mybrandnewdatatype );
MPI_Send ( myvec, 1, mybrandnewdatatype, ... );
```

Committing ad Freeing a Datatype

- **Before a dataytype handle is used in message passing communication, it needs to be committed with MPI_TYPE_COMMIT.**
- **This need be done only once (by each MPI process).**
- **If type no longer required, one may call MPI_TYPE_FREE() to free a datatype and its internal resources.**

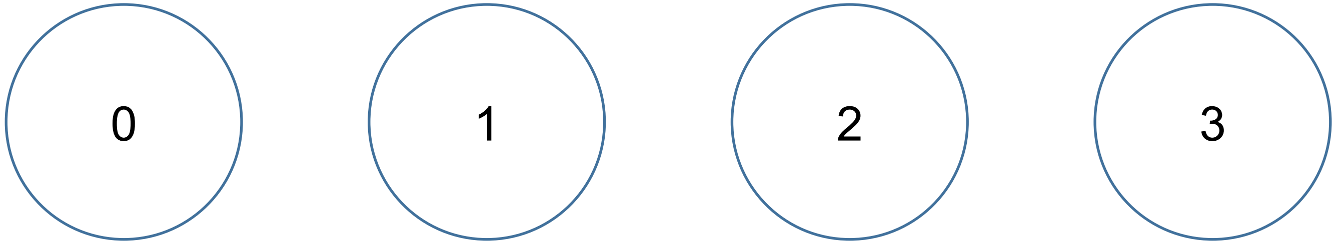
```
int MPI_Type_commit(MPI_Datatype *datatype);
int MPI_Type_free     (MPI_Datatype *datatype);
```

Outline

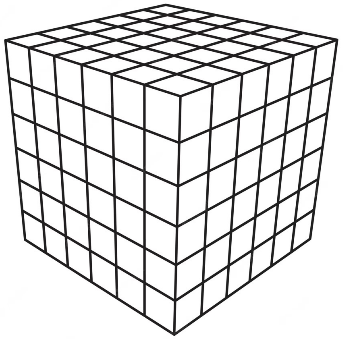
- Distributed Parallel Computing II
  - Non-Blocking Communication
  - Collective Communication
  - Derived Datatypes
  - **Virtual Topologies**
  - Hybrid MPI+OpenMP
- Quiz

Virtual Topologies

- Convenient process naming.
- Naming scheme to fit the communication pattern.
- Simplifies writing of code.
- Can allow MPI to optimize communications.
- Normal topology:



- Now consider that you work with this:

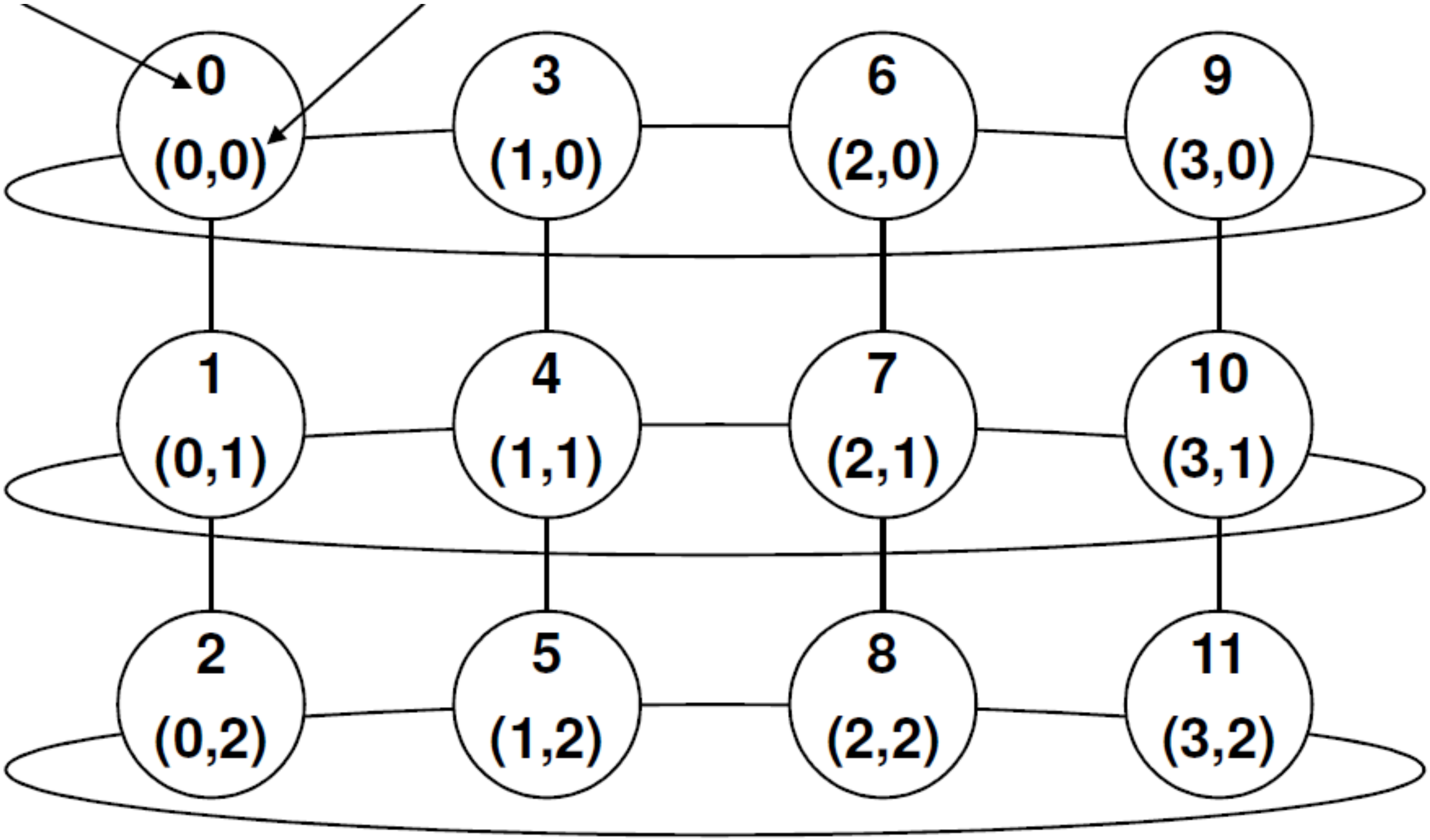


How to Use a Virtual Topology

- Creating a topology produces a new communicator.
- MPI provides mapping functions:
  - to compute process ranks, based on the topology naming scheme,
  - and vice versa.

Example – A Two-Dimensional Cylinder

- **Ranks** and **Cartesian process** coordinates



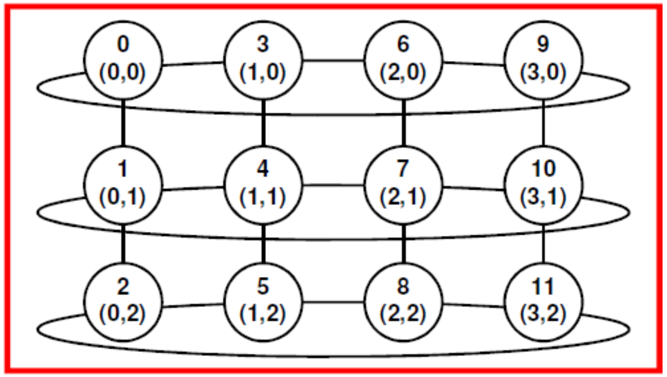
Topology Types

- Cartesian Topologies
  - each process is connected to its neighbor in a virtual grid,
  - boundaries can be cyclic, or not,
  - processes are identified by Cartesian coordinates,
  - of course, communication between any two processes is still allowed.
- Graph Topologies
  - general graphs,
  - two interfaces:
    - MPI_GRAPH_CREATE (since MPI-1)
    - MPI_DIST_GRAPH_CREATE_ADJACENT &
    - MPI_DIST_GRAPH_CREATE (new scalable interface since MPI-2.2)
  - not covered here.

Creating a Cartesian Virtual Topology

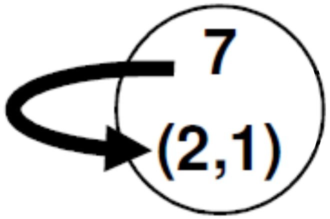
```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
                    int *dims, int *periods, int reorder,
                    MPI_Comm *comm_cart)
```

- **Comm_old = MPI_COMM_WORLD**
- **ndims = 2**
- **Dims = (4, 3)**
- **Periods = (1, 0)**
- **Reorder = 0 or 1**
  → Reorder: Task MPI backend to optimally assign MPI ranks to specific Cartesian coordinates considering communication ramifications!
  → But then ranks in comm_cart differ from comm_old!



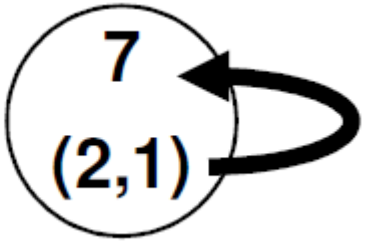
Cartesian Mapping Functions

- **Mapping ranks to process grid coordinates**



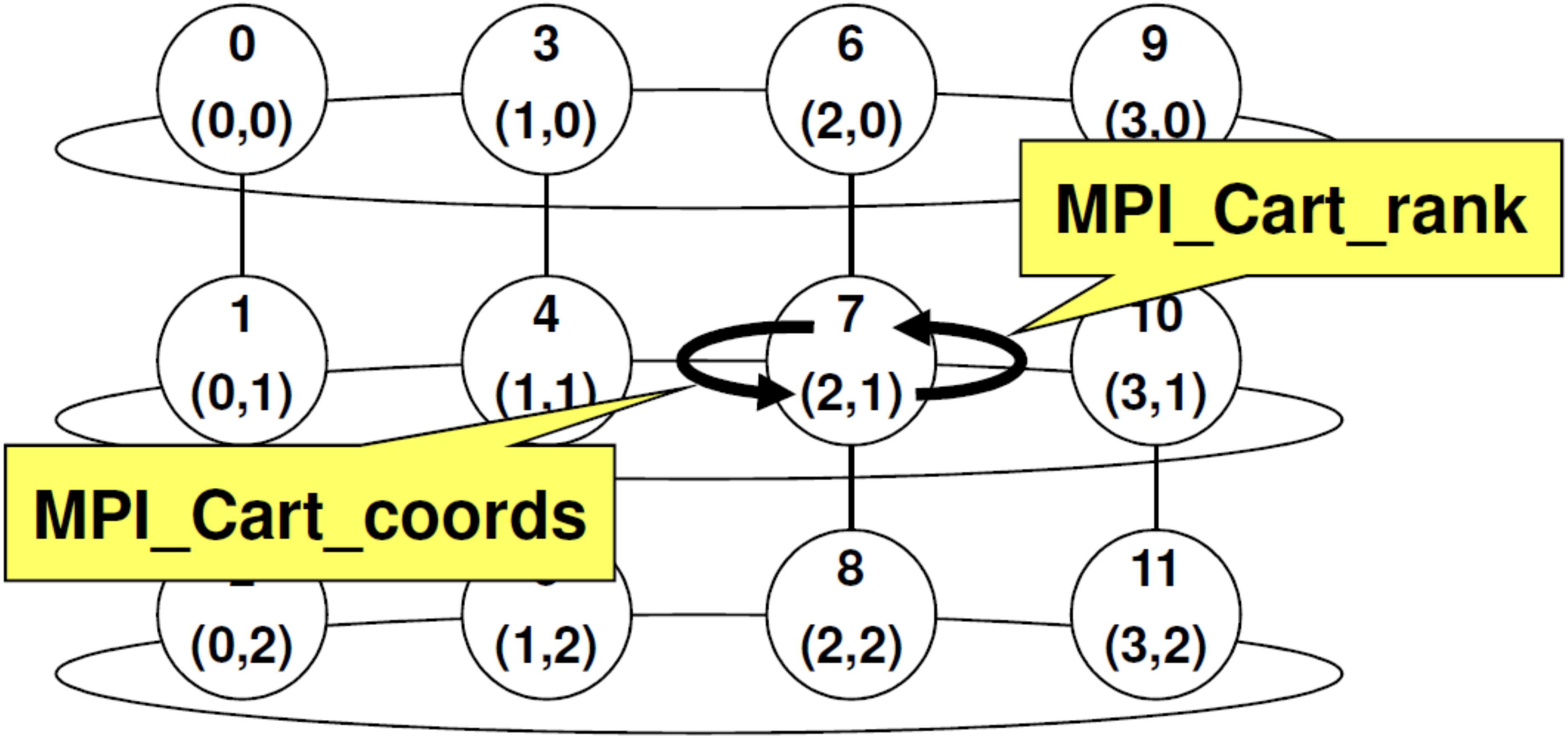

```
int MPI_Cart_coords(MPI_Comm comm_cart, int rank, int maxdims, int *coords)
```

- **Mapping process grid coordinates to ranks**




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

Cartesian Mapping Functions

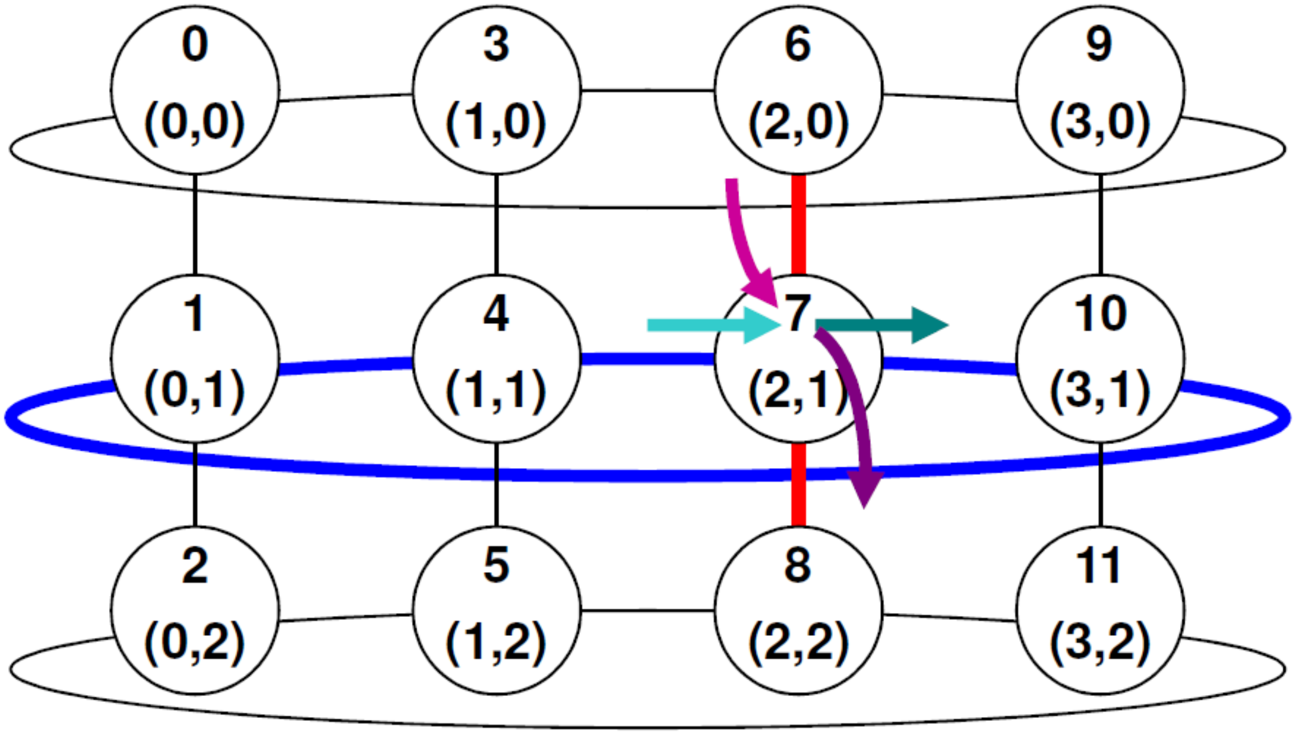


Cartesian Mapping Functions

- **Compute ranks of neighboring processes**
- **Returns MPI_PROC_NULL if there is no neighbor.**
- **In general: MPI_PROC_NULL can be used as source or destination rank in each communication. →
  This communication will be a no-operation!**

```
int MPI_Cart_shift(MPI_Comm comm_cart, int direction, int disp,
                   int *rank_source, int *rank_dest)
```

Cartesian Mapping Functions



invisible input argument: **my_rank** in cart

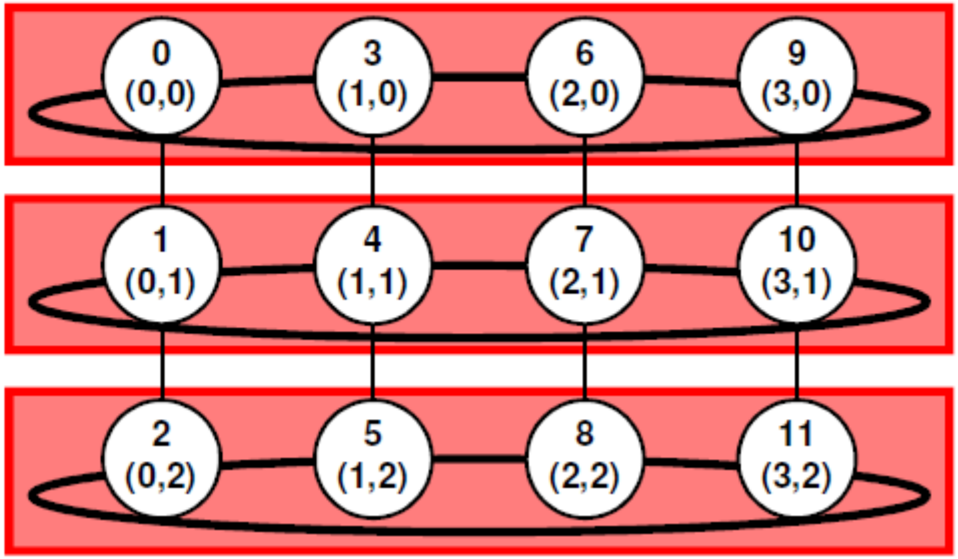
MPI_Cart_shift(cart, direction, displace, *rank_source*, *rank_dest*, *ierror*)

example on	0 or	+1	4	10
process rank=**7**	1	+1	6	8

Cartesian Partitioning

- **Cut a grid up into slices.**
- **A new communicator is produced for each slice.**
- **Each slice can then perform its own collective communications.**

```
int MPI_Cart_sub( MPI_Comm comm_cart, int *remain_dims,
                  MPI_Comm *comm_slice)
```

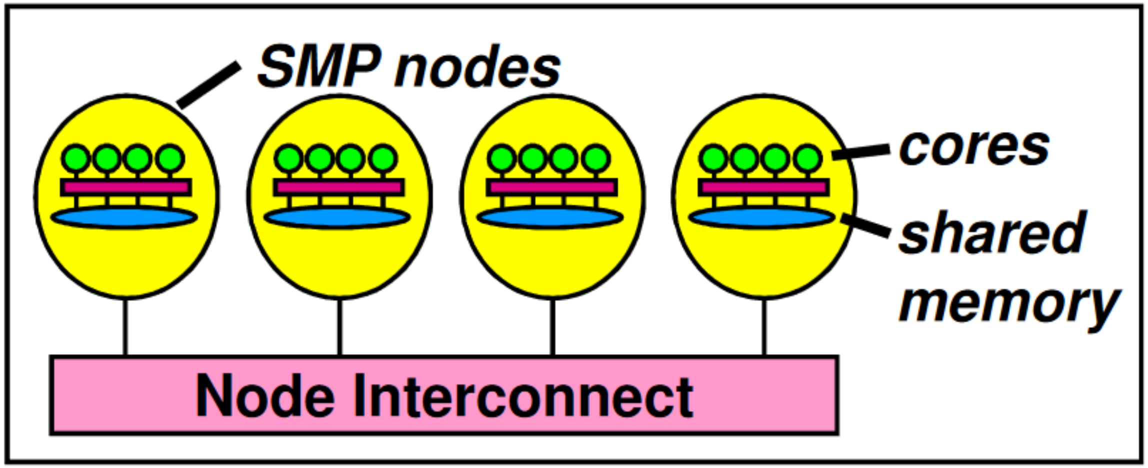


Outline

- Distributed Parallel Computing II
  - Non-Blocking Communication
  - Collective Communication
  - Derived Datatypes
  - Virtual Topologies
  - **Hybrid MPI+OpenMP**
- Quiz

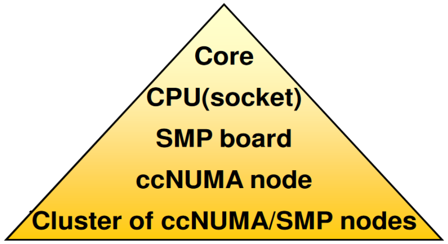
Hybrid Parallel Programming

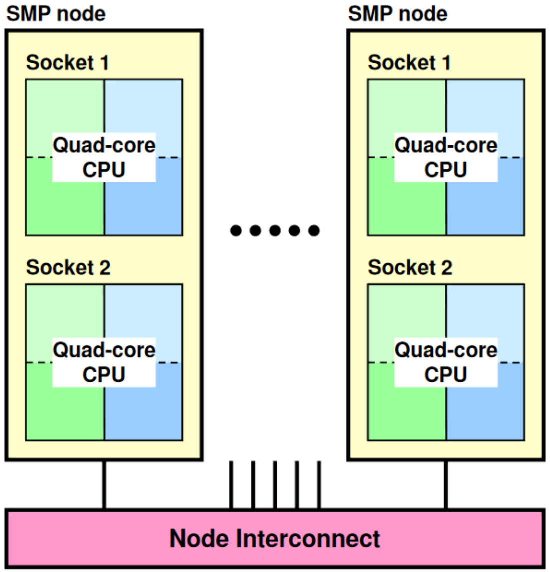
- **Hybrid parallel programming:**
  Mix different parallel programming approaches to efficiently use available computing platforms (heterogeneous computing resources)
- **Typical example:**
  Programming of clusters of shared memory nodes



Hybrid Parallel Programming

- Hierarchical system layout
- Hybrid programming seems *natural*
  - MPI between the nodes
  - Shared memory programming inside of each SMP node
    - OpenMP
    - MPI-3 shared-memory programming: not covered here
    - Accelerator programming (CUDA, OpenACC, etc): not covered here
    - And others.

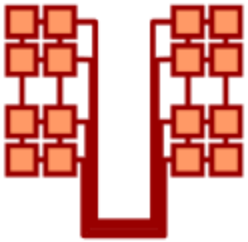




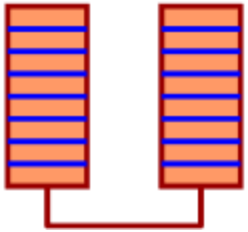
Hybrid Parallel Programming

- **Which programming model is best?**

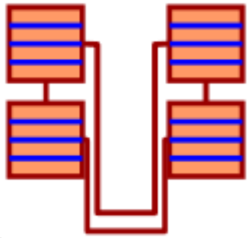
- **"Pure" MPI?**
  (a.k.a. 1 MPI process per CPU core,
  no shared-memory at all)



- **"Fully hybrid"?**
  (a.k.a. 1 MPI process per node,
  shared-memory within nodes)

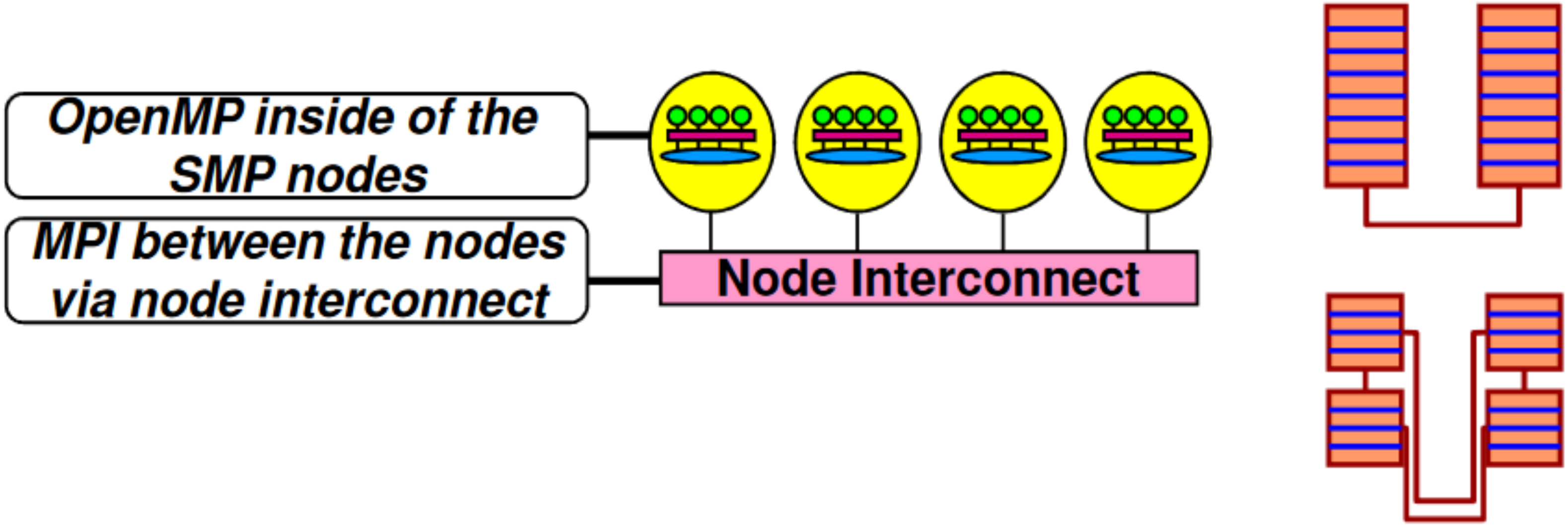


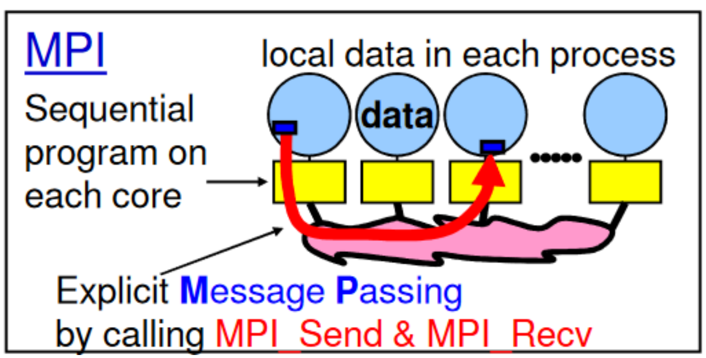
- **"Mixed hybrid"?**
  (a.k.a. 1 MPI process per, e.g., CPU,
  shared-memory within the CPUs;
  other combinations possible, e.g., (cc)NUMA specific)

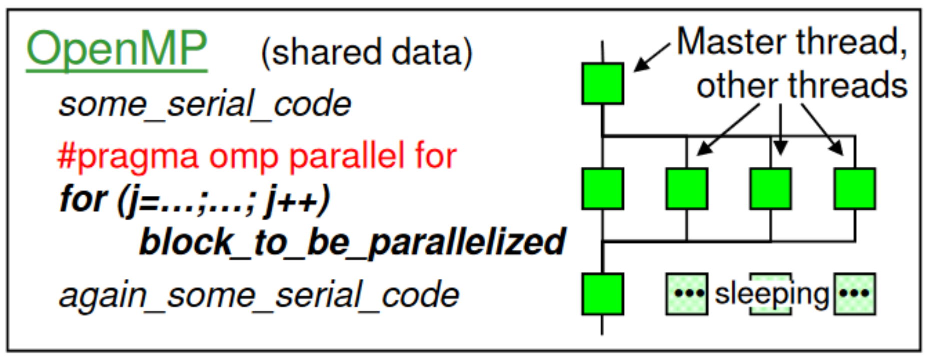


Problem specific …

Typical Case: MPI+OpenMP







Hybrid MPI+OpenMP *Masteronly*

- **Masteronly:**
  MPI only outside parallel OpenMP regions → only the master thread issues MPI calls

```
#pragma omp parallel
  numerical code
/*end omp parallel */

/* on master thread only */
MPI_Send (original data to halo areas in other SMP nodes)
MPI_Recv (halo data from the neighbors)
```

Hybrid MPI+OpenMP *Masteronly*

- **Major Advantages**
  - **No message passing inside of the SMP nodes**
  - **No topology problem**
- **Major Disadvantages**
  - **All other threads are sleeping while master thread communicates**
  - **All communicated data passes through the cache where the master thread is executing**
  - **Strictly speaking: MPI-library must have been compiled with threading support (configure OpenMPI library build with: `--enable-mpi-threads`)**
    and must have been initialized with threading support
    → But masteronly approach will likely work without as well.

MPI Threading Support Handles

- **MPI_THREAD_SINGLE: Only one thread will execute (→ similar to calling MPI_Init ())**
- **MPI_THREAD_FUNNELED: Only master thread will make MPI-calls**
- **MPI_THREAD_SERIALIZED: Multiple threads may make MPI-calls, but only one at a time (not covered here)**
- **MPI_THREAD_MULTIPLE: Multiple threads may call MPI, with no restrictions (not covered here)**

```
int MPI_Init_thread( int * argc, char ** argv[],
                     int thread_level_required,
                     int * thead_level_provided);
```

Outline

- Distributed Parallel Computing II
  - Non-Blocking Communication
  - Collective Communication
  - Derived Datatypes
  - Virtual Topologies
  - Hybrid MPI+OpenMP
- **Quiz**

Quiz

- **Q1: How many Bytes would be required to store an MPI derived datatype based on a struct of two characters and one double?**
- **Q2: Can a non-blocking receive be combined with a blocking send?**
- **Q3: Give an example setup for defining a Cartesian MPI communication topology for a three-dimensional setup if you could use up to 1000 MPI processes (make assumptions for all other properties).**
- **Q4: What is the order of the forward/backward Euler method?**
- **Q5: Are Runge-Kutta methods single-step or multi-step methods?**