MPI – Part II
Advanced Topics

More on Communication, Self-defined Datatypes, Self-defined Communicators, Load Balancing, Case Study: Game of Life

MPI – Part II - Contents

- **Different Types of Communication**
- **Complex Data Structures**
- **Self-defined Communicators**
- **Load Balancing**
- **Case Study: Game of Life**

Questions

- **Where is the data kept until it is received?**
- **When is a send complete?**

Blocking and Non-Blocking

- **Blocking:**
  - Relates to the completion of an operation in the sense, that used resources, i.e. buffers, are free to use again
- **Non-blocking:**
  - Functions return as soon as possible but provided buffers must not be touched until another appropriate call successfully indicates that they are not in use anymore
  - Even read-only access may be prohibited
  - Non-blocking communications are primarily used to overlap computation with communication to effect performance gains
- **Blocking sends can be combined with non-blocking receives and vice versa.**

Completion of Non-Blocking Operations

- **In non-blocking send-variants we need to check for the communication's completion**

- **There are two options in checking for a communication's completion:**
  - Wait until the communication is complete using `MPI_Wait`
  - Loop with test until communication is completed using `MPI_Test`

- **To track communication requests an integer request handle is provided by the MPI system, e.g.**
  - `int MPI_Isend(... like MPI_Send, MPI_Request *req)`
  - `int MPI_Wait(MPI_Request *req, MPI_Status *status)`

Example: Send and wait

```
[…]
        message=42;
        MPI_Request req;
        MPI_Issend(&message, 1, MPI_INT, 1, tagSend, MPI_COMM_WORLD,
                &req);

        int flag=0;
        while (1)
        {
                MPI_Test(&req,&flag,MPI_STATUS_IGNORE);

                if (1 == flag)
                        break;

                printf("wait ...\n");
                sleep(1);
        }

        printf("Message sent \n");
[…]
```

Example: Receive and wait

```
[…]
        MPI_Request req;
        MPI_Irecv(&message, 1, MPI_INT, 0, tagSend, MPI_COMM_WORLD,
                  &req); // no status

        int flag=0;
        while (1)
        {
                MPI_Test(&req,&flag,MPI_STATUS_IGNORE);

                if (1 == flag)
                        break;

                printf("wait ...\n");
                sleep(1);
        }

        printf("Recv. Message: %i\n",message);
[…]
```

- **Relation between Sender and Receiver**

- **Synchronous:**
  - send call will only start when the destination has started synchronous receive
  - send operation will complete only after acknowledgement that the message was safely received by the receiving process (destination has copied data out of incoming buffer into memory)

- **Asynchronous (buffered):**
  - a send operation may "complete" even though the receiving process has not actually received the message
  - only know that message has left

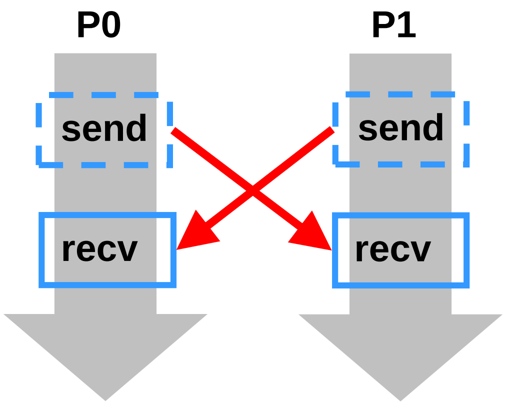
Variants of Communication

semantics / mode	standard	synchronous	Asynchronous (buffered)
blocking	MPI_Send MPI_Recv	MPI_Ssend	MPI_Bsend
non-blocking	MPI_Isend MPI_Irecv	MPI_Issend	MPI_Ibsend

- **Note: There is only one receive function for both blocking and non-blocking send functions**

Deadlocks with MPI_Send

- `MPI_Send` can be synchronous, asynchronous or both (not declared in the MPI standard)
- Behavior is implementation dependent (typical: asynchronous for small messages and synchronous for large messages)
- Depending on the system, this can deadlock or not:



Example: Deadlock

```
[…]
        // force sync. send, wait for rcv. in any case
        // MPI_Send(&messageS, 1, MPI_INT, 1-procRank, tagSend, MPI_COMM_WORLD);
        MPI_Ssend(&messageS, 1, MPI_INT, 1-procRank, tagSend, MPI_COMM_WORLD);
        MPI_Recv(&messageR, 1, MPI_INT, 1-procRank, tagSend, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);

        printf("proc %d finished, message %d \n",procRank,messageR);
[…]
```

Example: No Deadlock

```
[…]
        // force sync. send, wait for rcv. in any case
        // MPI_Send(&messageS, 1, MPI_INT, 1-procRank, tagSend, MPI_COMM_WORLD);
        MPI_Bsend(&messageS, 1, MPI_INT, 1-procRank, tagSend, MPI_COMM_WORLD);
        MPI_Recv(&messageR, 1, MPI_INT, 1-procRank, tagSend, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);

        printf("proc %d finished, message %d \n",procRank,messageR);
[…]
```

Send and Receive in one Operation (1)

- `MPI_Sendrecv` combines an asynchronous send and receive
- Send buffer and receive buffer must be disjoint, and may have different lengths and data types
- A message sent by a send-receive operation can be received by a regular receive operation or probed by a probe operation
- A send-receive operation can receive a message sent by a regular send operation
- Useful for data exchange

Send and Receive in one Operation (2)

- **Send and receive buffer must not overlap**

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

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

Example: Sendreceive

```
[…]
        if (0 == procRank)
        {
                messageMaster = 42;

                MPI_Sendrecv(&messageMaster, 1, MPI_INT, 1, tagSendMaster,
                        &messageWorker, 1, MPI_INT, 1, tagSendWorker,
                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        else
        {
                messageWorker = 43;

                MPI_Sendrecv(&messageWorker, 1, MPI_INT, 0, tagSendWorker,
                        &messageMaster, 1, MPI_INT, 0, tagSendMaster,
                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

        printf("proc %d, master %d, worker %d \n",
                procRank, messageMaster, messageWorker);
[…]
```

Example: Sendreceive Replace

```
[…]
        if (0 == procRank)
        {
                message = 42;

                MPI_Sendrecv_replace(&message, 1, MPI_INT, 1, tagSendMaster,
                        1, tagSendWorker, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        else
        {
                message = 43;

                MPI_Sendrecv_replace(&message, 1, MPI_INT, 0, tagSendWorker,
                        0, tagSendMaster, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

        printf("proc %d, message %d \n",procRank, message);
[…]
```

Test for Incoming Messages

- **Also possible in a non-blocking mode:**
  - `int MPI_Iprobe(int src, int tag, MPI_Comm comm, int* flag, MPI_Status* status)`

  - flag: non-zero, if there is a matching message
  - status: source, tag and size of the message

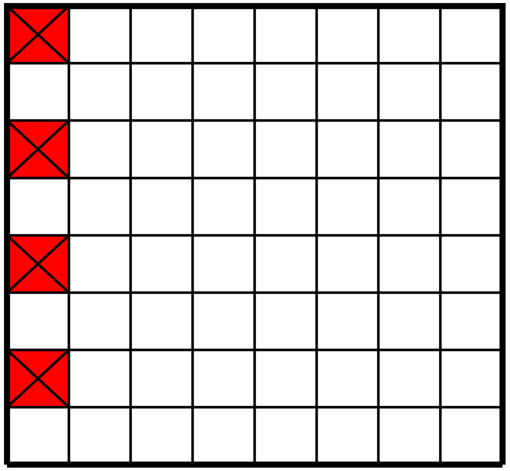
  - Not necessary to receive the message immediately after it has been probed for
  - The same message may be probed for several times before it is received

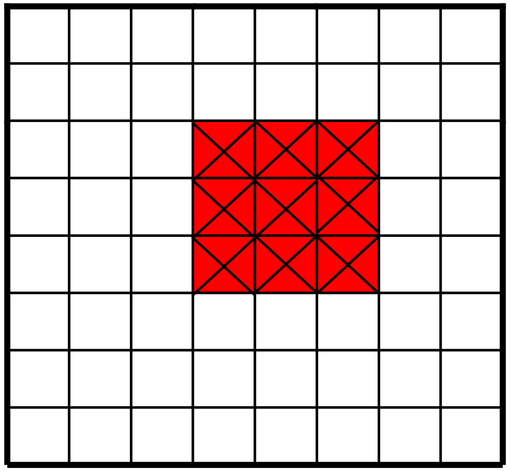
15. Test-Receive the measurement

MPI – Part II - Contents

Motivation

- **Using predefined data types would mean to send very small messages many times (e.g. communication of sub arrays)**





Derived Data Types

- **Data types in MPI: basic (already known) and derived**
- **MPI provides data type constructor functions to create derived data types**
- **Kinds of data type constructors in MPI:**
  - **contiguous**
  - **vector**/hvector
  - indexed/hindexed
  - struct

Committing and Freeing a Data Type

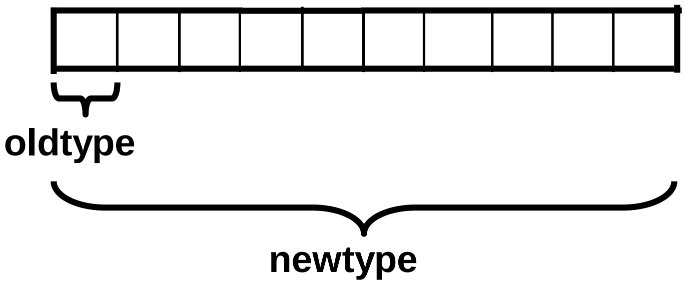
- **Before a data type handle is used in communication, it needs to be committed**
  - `int MPI_Type_commit(MPI_Datatype *datatype)`

- **After use, a self defined data type can be deallocated**
  - `int MPI_Type_free(MPI_Datatype *datatype)`

Contiguous Data Type

- Simplest derived data type

- Creates a new data type consisting of contiguous elements of another data type




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

Transfer of Contiguous Data

- **“New” method (derived data type):**

count elements of old type

```
❖ MPI_Type_contiguous(count, datatype, &newtype)
  MPI_Type_commit(&newtype)
  MPI_Send(&buffer, 1, newtype, dest, tag, comm)
```

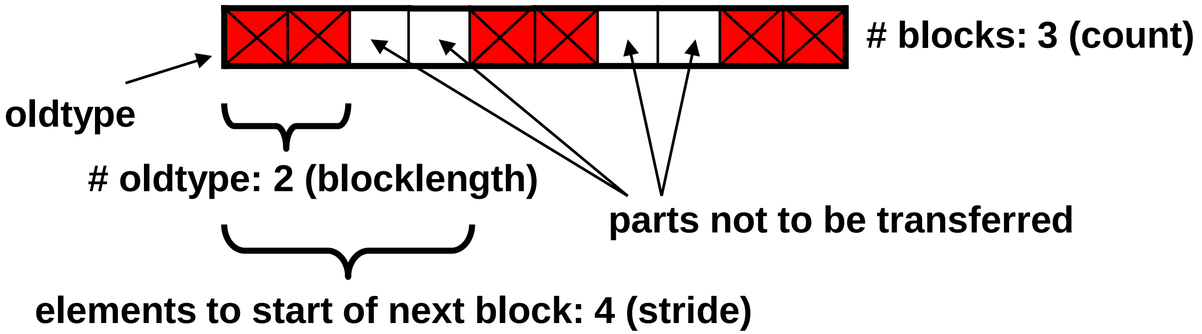
- **“Old” method (simple data type)**

```
❖ MPI_Send(&buffer, count, datatype, dest, tag, comm)
```

count elements of old type

Vector Data Type

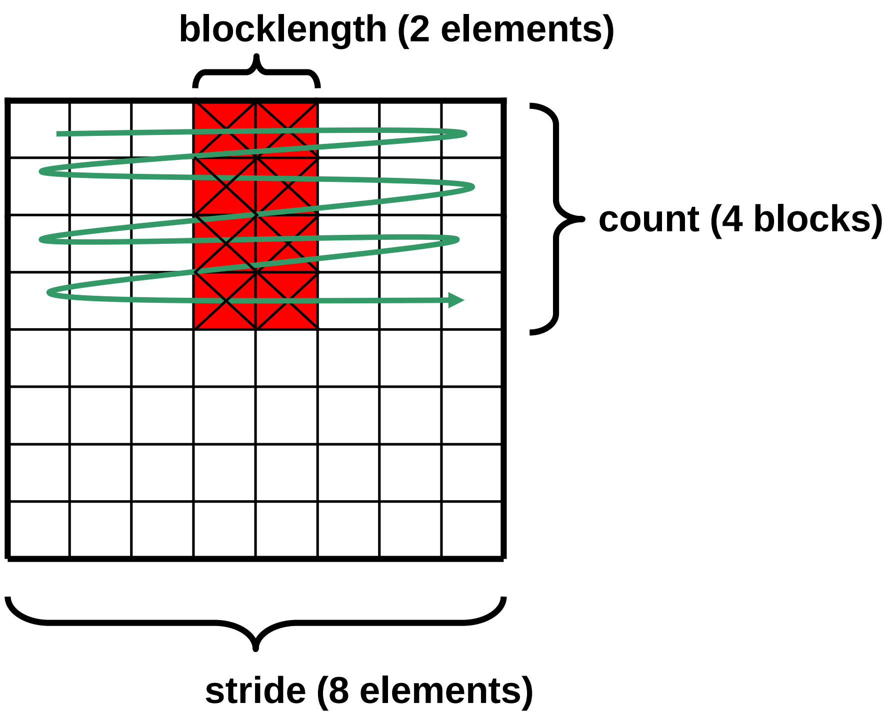
- More general constructor
- Allows replication of a data type into locations that consist of equally spaced blocks




```
int MPI_Type_vector(int count, int blocklength,
int stride, MPI_Datatype oldtype, MPI_Datatype
*newtype)
```

Example: Transfer of 2D Array Data

- `MPI_Type_vector` is fine for sending rectangular blocks from 2D arrays



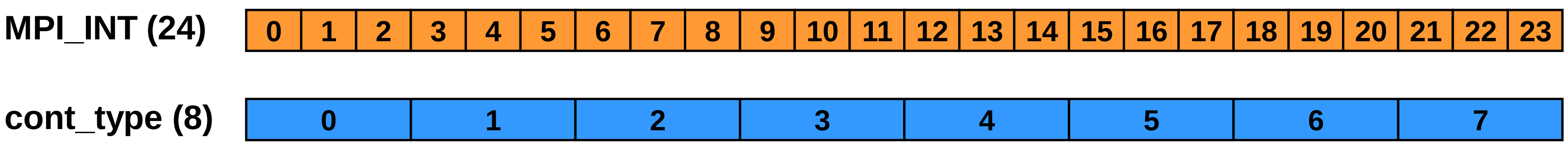
Note: in Fortran: different memory layout

Example: Contiguous Data (1 / 2)

```
[…]
        MPI_Type_contiguous(3, MPI_INT, &cont_type); // make type: 24 = 8*3
        MPI_Type_commit(&cont_type);

        if (0 == procRank)
        {
                for (int i=0; i<24; i++)
                        buffer[i] = i;

                // send data with new data type to worker
                MPI_Send(buffer, 8, cont_type, 1,  tagSendBuffer, MPI_COMM_WORLD);
        }
        else
        {
                // receive data with new data type from master
                MPI_Recv(buffer, 8, cont_type, 0, tagSendBuffer, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);

                for (int i=0; i<24; i++)
                        printf("buffer[%d] = %d\n", i, buffer[i]);
        }
[…]
```

Example: Contiguous Data (2 / 2)

- **The contiguous data type created in the source code on the slide before looks like this:**



Example: Vector Data of Basic Type (1 / 2)

```
[…]
        MPI_Type_vector(3, 6, 9, MPI_INT, &vec_type); // 3 blocks, length 6, stride 9
        MPI_Type_commit(&vec_type);

        if (0 == procRank)
        {
                for (int i=0; i<24; i++)
                        buffer[i] = 2*i;

                // send data with new data type to worker
                MPI_Send(buffer, 1, vec_type, 1,  tagSendBuffer, MPI_COMM_WORLD);
        }
        else
        {
                for (int i=0; i<24; i++)
                        buffer[i] = -1;

                // receive data with new data type from master
                MPI_Recv(buffer, 1, vec_type, 0, tagSendBuffer, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);

                for (int i=0; i<24; i++)
                        printf("buffer[%d] = %d\n", i, buffer[i]);
        }
[…]
```

Example: Vector Data of Basic Type (2 / 2)

- **The vector data type based on a basic data type created in the source code on the slide before looks like this:**

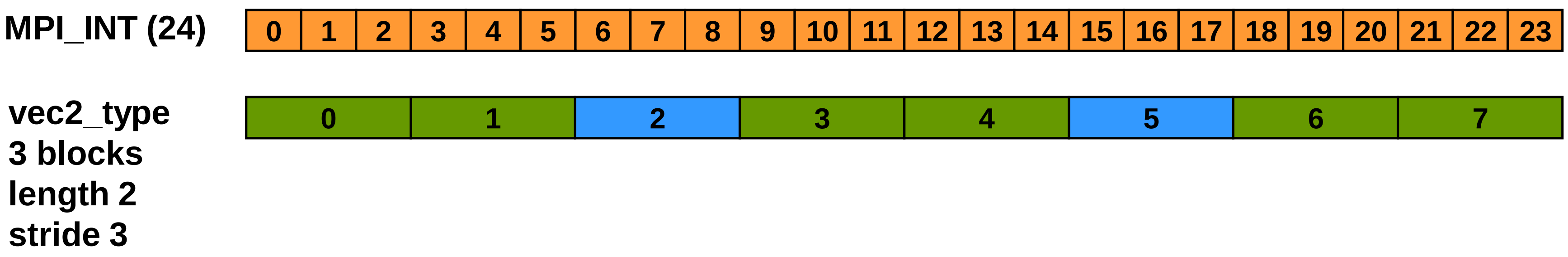
MPI_INT (24)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
vec_type **3 blocks** **length 6** **stride 9**	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Example: Vector Data of own Data Type (1 / 2)

```
[…]
        MPI_Type_vector(3, 2, 3, cont_type, &vec2_type); //3 blocks, length 2, stride 3
        MPI_Type_commit(&vec2_type);

        if (0 == procRank)
        {
                for (int i=0; i<24; i++)
                        buffer[i] = 3*i;

                // send data with new data type to worker
                MPI_Send(buffer, 1, vec2_type, 1,  tagSendBuffer, MPI_COMM_WORLD);
        }
        else
        {
                for (int i=0; i<24; i++)
                        buffer[i] = -1;

                // receive data with new data type from master
                MPI_Recv(buffer, 1, vec2_type, 0, tagSendBuffer, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);

                for (int i=0; i<24; i++)
                        printf("buffer[%d] = %d\n", i, buffer[i]);
        }
[…]
```

Example: Vector Data of own Data Type (2 / 2)

- **The vector data type based on a self-defined data type created in the source code on the slide before looks like this:**



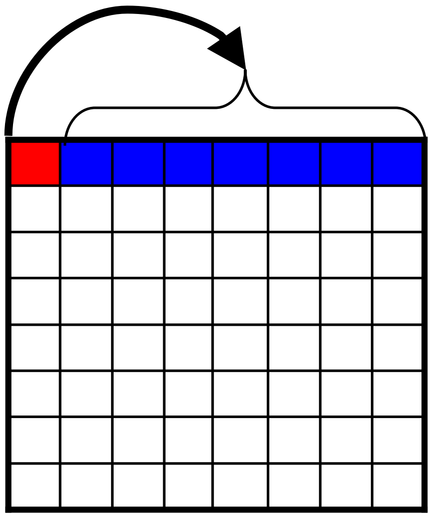
16. Vector data type

MPI – Part II - Contents

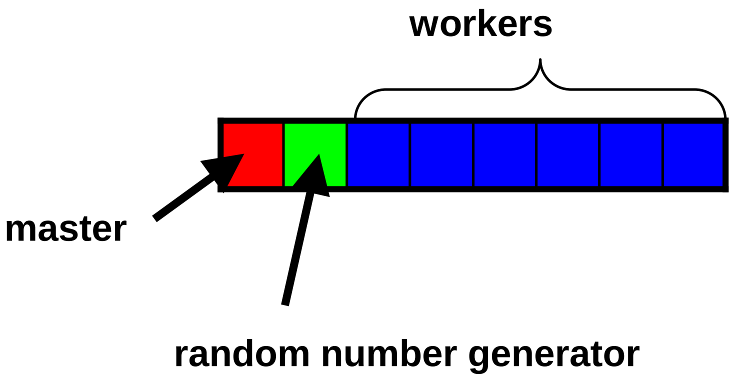
- Different Types of Communication
- Complex Data Structures
- **Self-defined Communicators**
- Load Balancing
- Case Study: Game of Life

Why new Communicators?

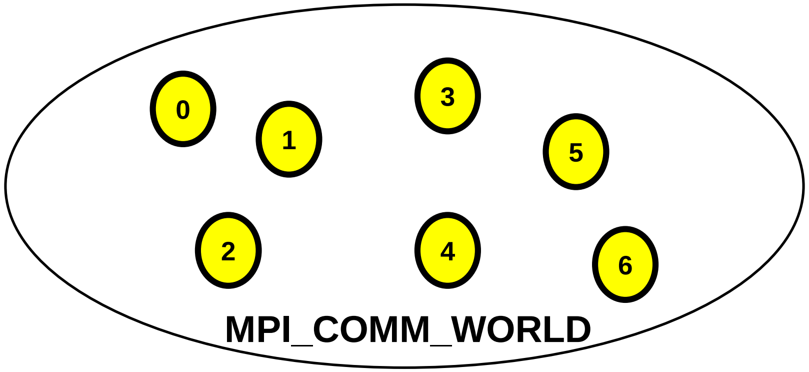
- **Communication in a single row of processes instead of the whole matrix**



- **Different tasks for processes**



- **Base communicator for all MPI communicators is predefined: `MPI_COMM_WORLD`**



- **One can**
  - either construct a new communicator (means to form a new group)

  or

  - split a communicator into a group of new communicators

Construct new Communicator

- **Extract the "process group" from the communicator with**
  - `int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)`
- **Use MPI_Group-Commands to alter groups**
  - `int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)`
  - `int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)`
- **Create a communicator around a new group with**
  - `int MPI_Comm_create(MPI_Comm oldcomm, MPI_Group newgroup, MPI_Comm *newcomm)`

  collective command - processes of the old communicator not included get a dummy value

Example: Construct new Communicator

```
[…]
        // remove two processes
        int loser[2]; //have to leave the world_group
        MPI_Group world_group, win_group;
        MPI_Comm win_comm;

        // first and last have to go
        loser[0]=0;
        loser[1]=procCount-1;

        // return group of communicator
        MPI_Comm_group(MPI_COMM_WORLD, &world_group);

        // create new group without loser
        MPI_Group_excl(world_group, 2, loser, &win_group);

        // create communicator (subset of group of comm)
        MPI_Comm_create(MPI_COMM_WORLD, win_group, &win_comm);
[…]
```

Split a Communicator

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

- **A new communicator *newcomm* is created for each value of color**

- **Color value `MPI_UNDEFINED` allowed, in which case newcomm returns `MPI_COMM_NULL`**

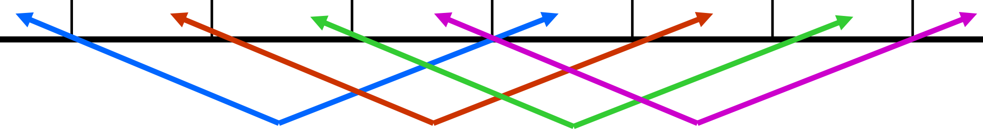
- **Within the new communicators, the processes are ranked in the order(!) defined by the value of the argument key**

- **Collective call, but each process can provide different values for color and key**

Example: Split in 4 Groups (1 / 2)

```
[…]
        int color = procRank%4;

        MPI_Comm newcomm;
        MPI_Comm_split(MPI_COMM_WORLD, color, procRank,
                        &newcomm);
[…]
```

rank (=key)	0	1	2	3	4	5	6	7
color	0	1	2	3	0	1	2	3



4 new communicators

new comm	0	1	2	3	0	1	2	3
new rank	0	0	0	0	1	1	1	1

Communicator Destructor

- `int MPI_Comm_free(MPI_Comm *comm)`

- **Collective operation**

- **Marks the communication object for deallocation**

- **Any pending operations that use this communicator will complete normally**

- **The object is actually deallocated only if there are no other active references to it**

17. Define Communicators

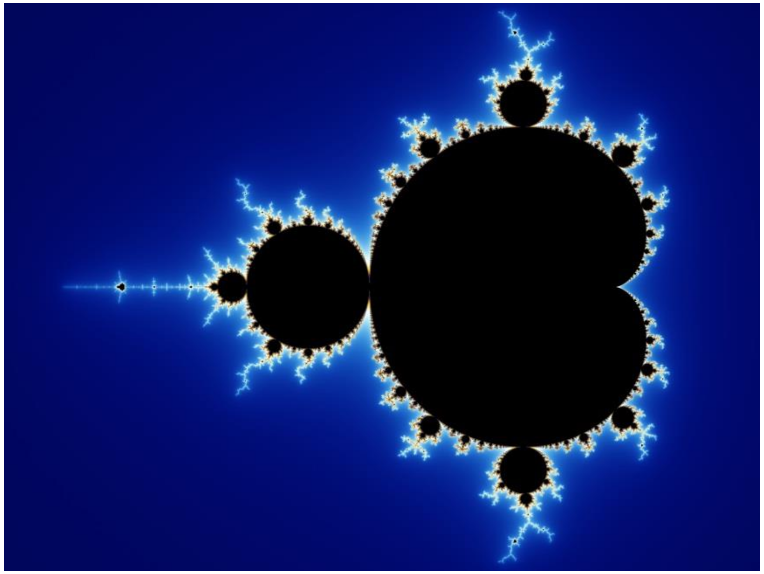
MPI – Part II - Contents

- Different Types of Communication
- Complex Data Structures
- Self-defined Communicators
- **Load Balancing**
- Case Study: Game of Life

Example: Mandelbrot Set

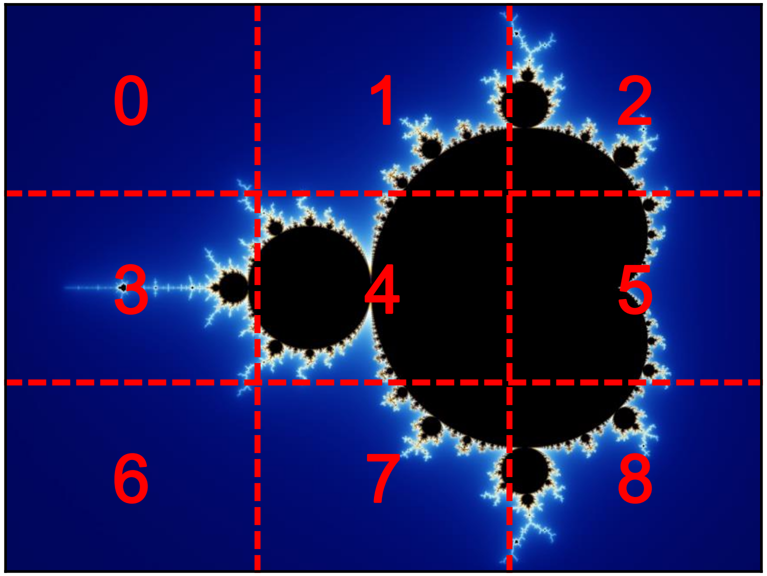
- **The Mandelbrot set is a set of complex values**

- **A complex number c is in the Mandelbrot set if, when starting with $x_0 = 0$ and applying the iteration $x_{n+1} = x_n^2 + c$ repeatedly, the absolute value of x_n never exceeds a certain number**



Parallel Calculation of the Set

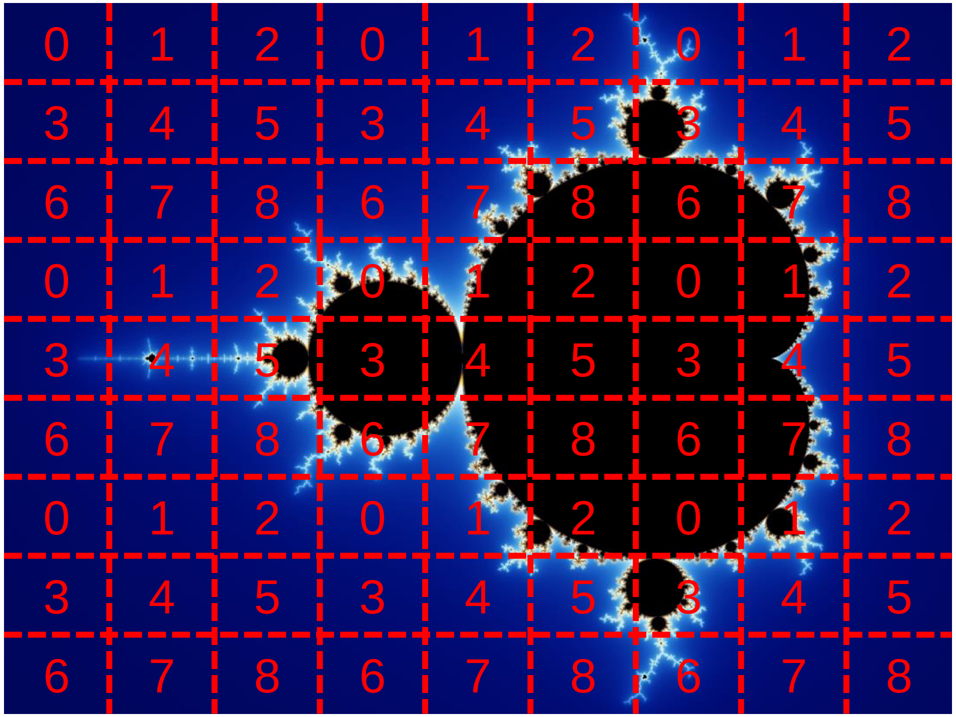
- Each process calculates one part of the area
- Process 4 has much more work to do than process 0 and 6



⇨ **Load balancing problem**

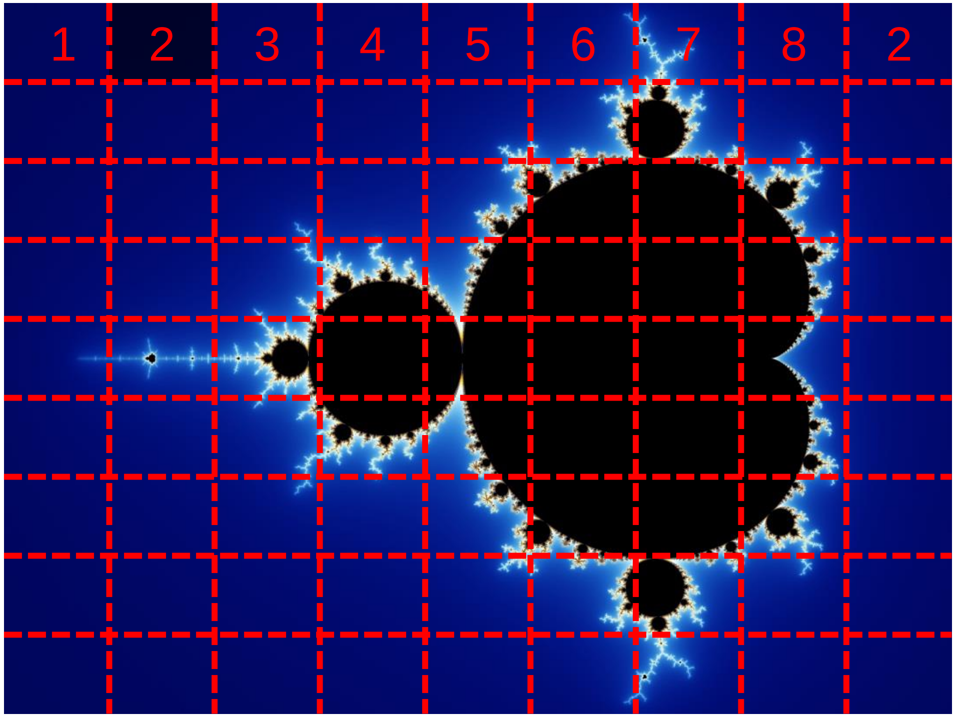
Solution #1

- **Redesign the distribution of your processes**
- **The over-all workload is nearly equal for all processes**
- **The distribution can also be determined by random numbers**



Master-Worker-Pattern: First Result

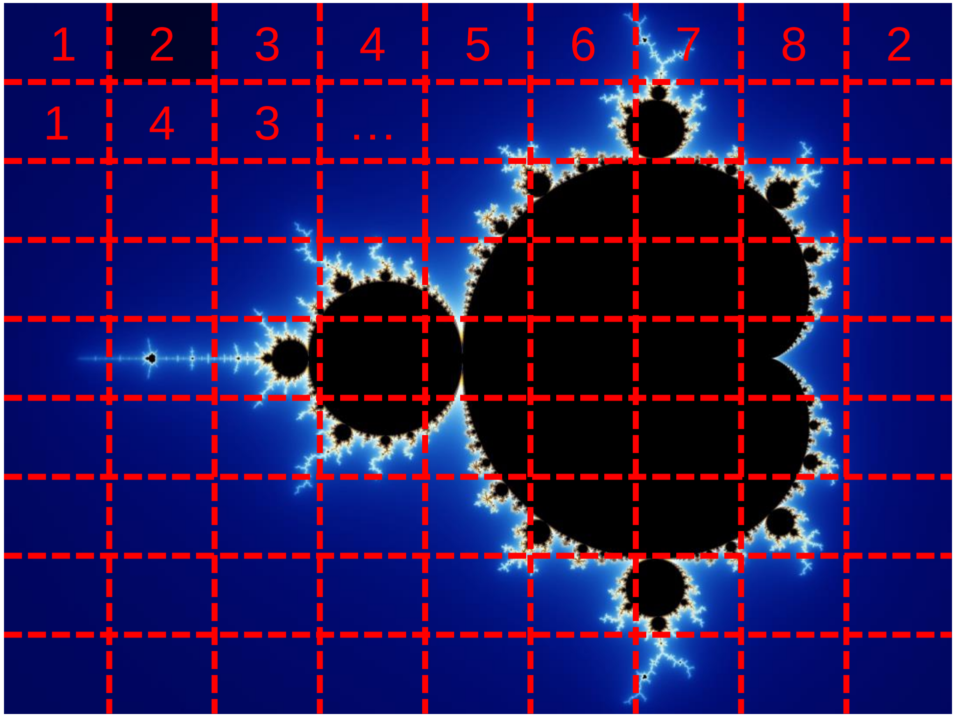
- **Assume process 2 is ready first**
  - gives the results to the master and asks for more work
  - process 2 gets the next free block
  - and so on …



Master-Worker-Pattern: Last Results

- **Now, let any process be ready:**
  - no more blocks to calculate
  - master sends a message saying: finish your work (e.g. with a special tag)
  - when all workers have finished their work, the master finishes, too
  - **`MPI_Finalize`** is a collective operation

  ⇨ the program ends, when all processes have called it

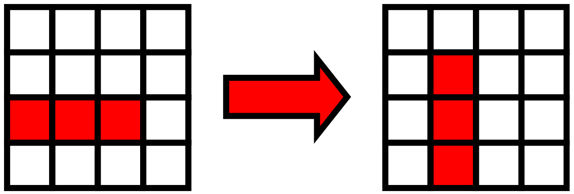


MPI – Part II - Contents

- **“cellular automaton”**
  - iterations over a 2d-array
  - cells can be live (1) or dead (0)
  - each cell interacts with its 8 neighbors

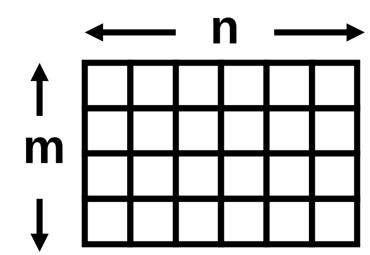
0	0	0
1	1	0
0	1	0

- **Rules for each iteration (all cells at the same time!):**
  - 0-1 neighbors live: new state=0 (living cells die)
  - 2 neighbors live: new state=old state
  - 3 neighbors live: new state=1 (dead cells come to live)
  - 4-8 neighbors live: new state=0 (living cells die)
  - Example:



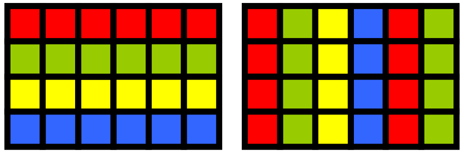
Domain Decomposition (1 / 2)

- **game of life takes place on a m x n grid**

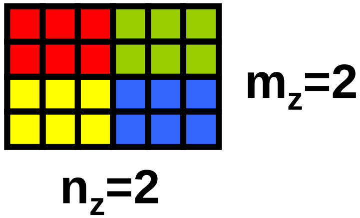


- **distribute the grid on z processors (domain decomposition)**
  - simplest way: row wise or column wise
  - more general approach: rectangular areas

    (checkerboard partitioning)



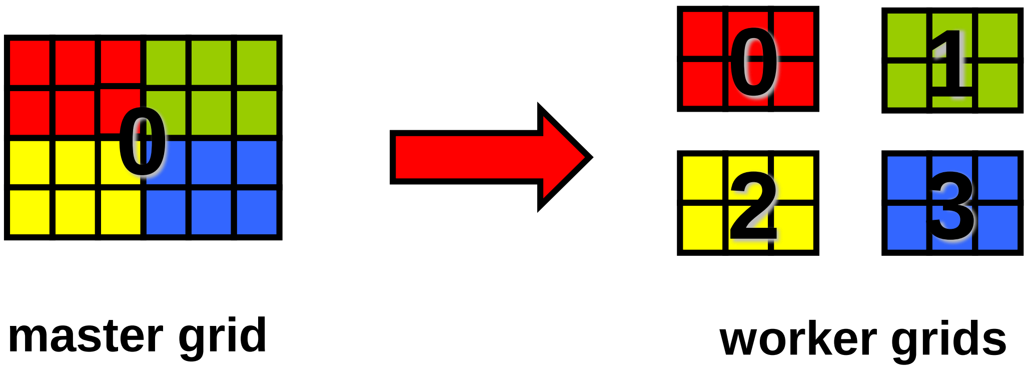
  - constraints:
    - $m \,\%\, m_z = 0$
    - $n \,\%\, n_z = 0$
    - $m_z \cdot n_z = z$



  - get the best (most compact) distribution with `MPI_Dims_Create`

Domain Decomposition (2 / 2)

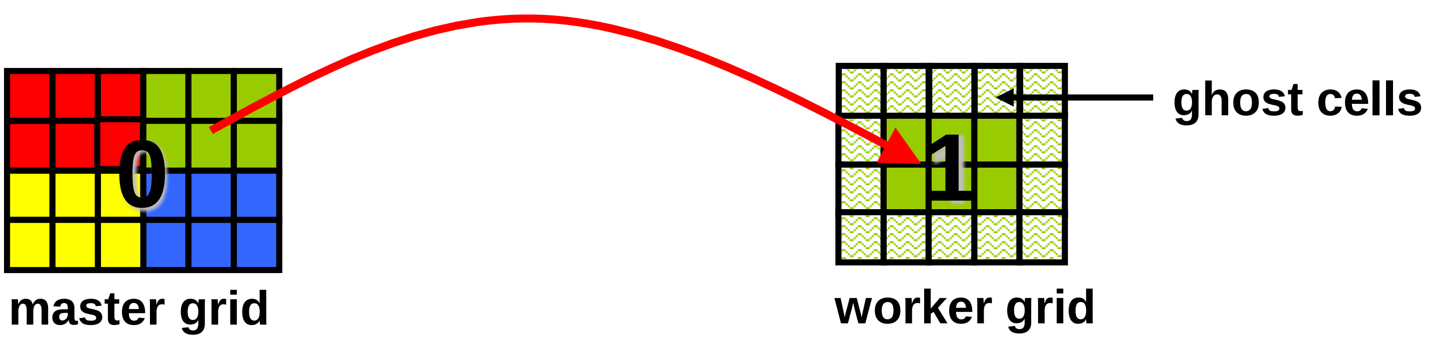
- Initial (master) grid is in process 0
- Parts must get distributed to the other processes



Ghost Cells

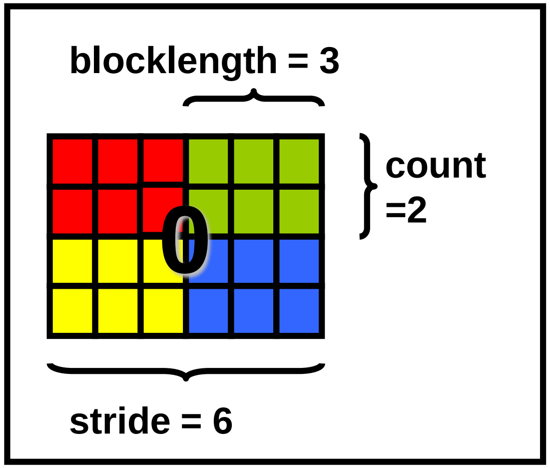
- **For updating the cells, we need all the neighbours of all the cells**
  - "ghost cells" around each block are necessary

- **This means**
  - cells are not continuous in memory; neither in the master nor in the worker grid
  - they are arranged in different ways

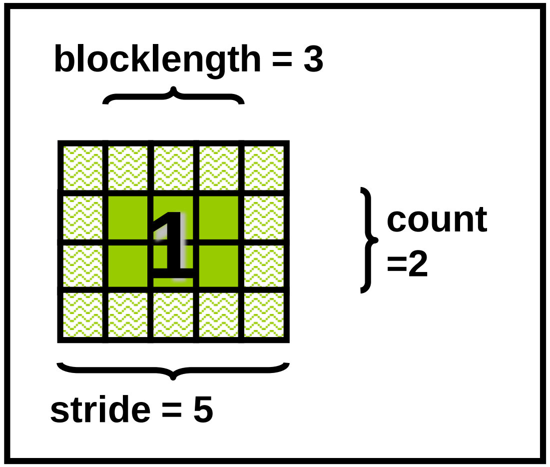


- **Define new MPI datatypes with**
  - `MPI_Type_vector(count, blocklength, stride, oldtype, &newtype)`

master_type



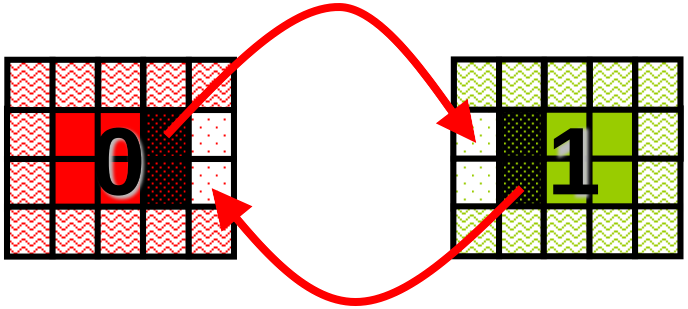
worker_type



- **The “type signature” of both types matches**
- **It is possible to send `master_type` and receive `worker_type`**

Data Exchange

- **Exchange between subdomains to fill the ghost cells**



- **Each process exchanges borders with all its 4 neighbours**

- **Use again new MPI data types:**
  - rows: `MPI_Type_contiguous`
  - columns: `MPI_Type_vector`

- **Data exchange with `MPI_Sendrecv`**

Iterations and Visualization

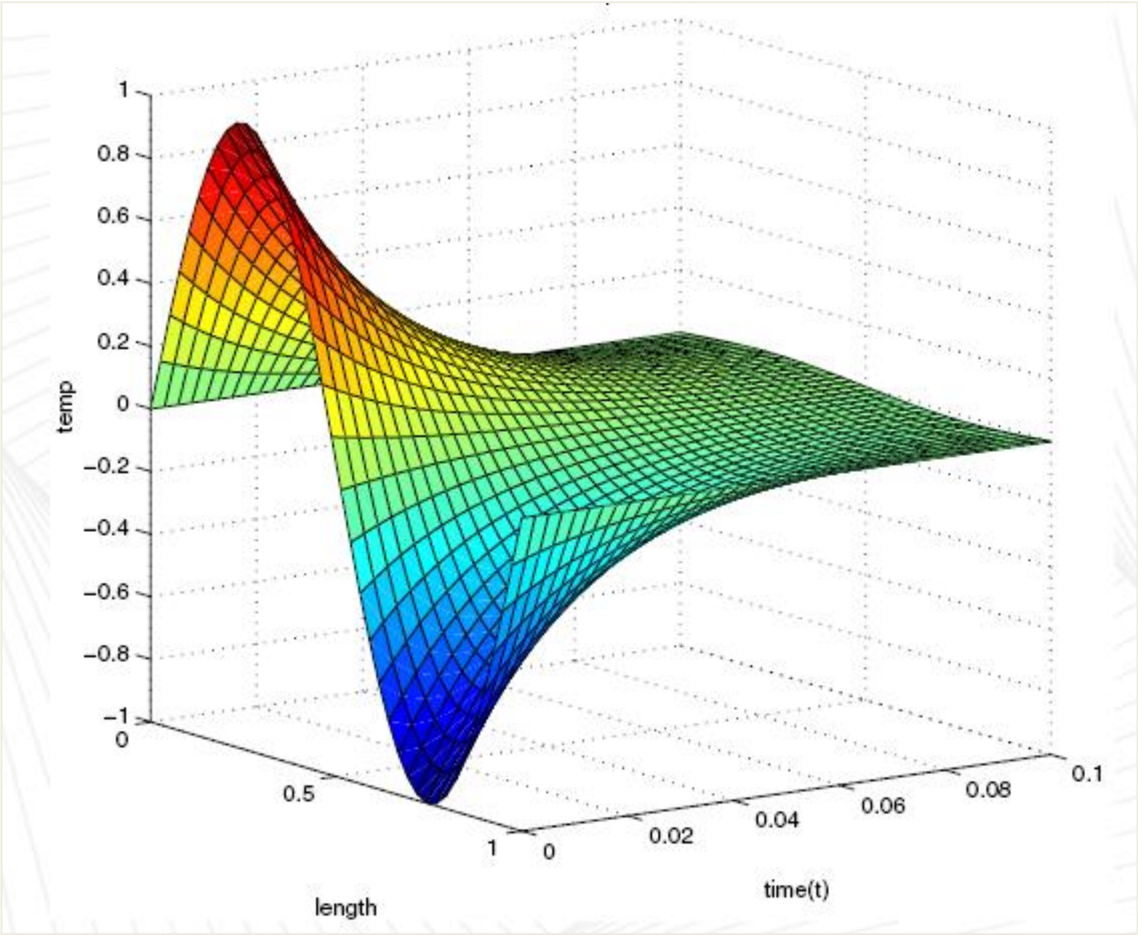
- **The calculation is a sequence of iterations and data exchanges between subdomains**
- **When the data is to be visualized, all the worker data must be copied back to the master grid**
- **More about GOL visualization …**
  http://golly.sourceforge.net

Summary of Part II

- **Blocking/non-blocking , synchronous/asynchronous**
- **Derived data types**
- **New communicators**
- **Load balancing**

Example / Exercise: Heat-Diffusion

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2}$$



$$\frac{u_i^{(j+1)} - u_i^{(j)}}{\Delta t} = \frac{u_{i+1}^{(j)} - 2u_i^{(j)} + u_{i-1}^{(j)}}{(\Delta x)^2}$$

Exercises

MPI

Exercise 15 – Measure Test-Receive Speed

- **Count how often you may call MPI_Test (polling) in one second waiting for a message:**
  - Use two processes. One sends a message the other waits for the message in non-blocking mode.

Exercise 16 – Vector Datatype

Rewrite your solution of Exercise 13 using self-defined datatypes:

- Define a MPI-datatype `row_type´ that represents a complete row in A.

- Send elements of this `row_type´ instead of doubles, i.e. use this type for scatter and gather operations.

0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	0
2	3	4	5	6	7	8	9	0	1
3	4	5	6	7	8	9	0	1	2
4	5	6	7	8	9	0	1	2	3
5	6	7	8	9	0	1	2	3	4
6	7	8	9	0	1	2	3	4	5
7	8	9	0	1	2	3	4	5	6

One element of type `row_type´!

Exercise 17 - Communicator

- **Define Communicators:**
  - Define a communicator with exactly 20 processes and divide them into groups of 5 (the first communicator contains (old) rank 0..4, etc.)
  - Start your program with 25 processes.

Exercise 18 – Heat Equation (1 / 10)

- The heat equation is a so called partial differential equation describing the distribution of heat or variation in temperature in a given region over time. It reads

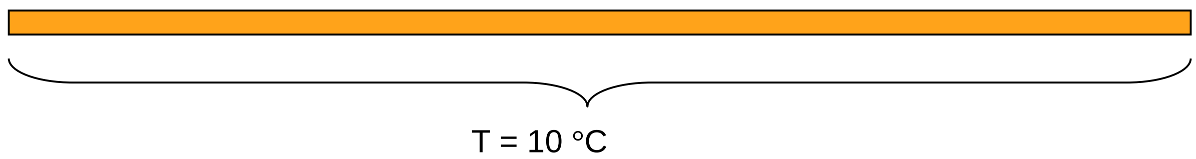
$$u_t = k^2 u_{xx}$$

- Here k denotes some material constant (set $k = 1.0$ here) and $u(x, t)$ a function of one spatial variable x and time variable t, representing the temperature at a point x at time t.

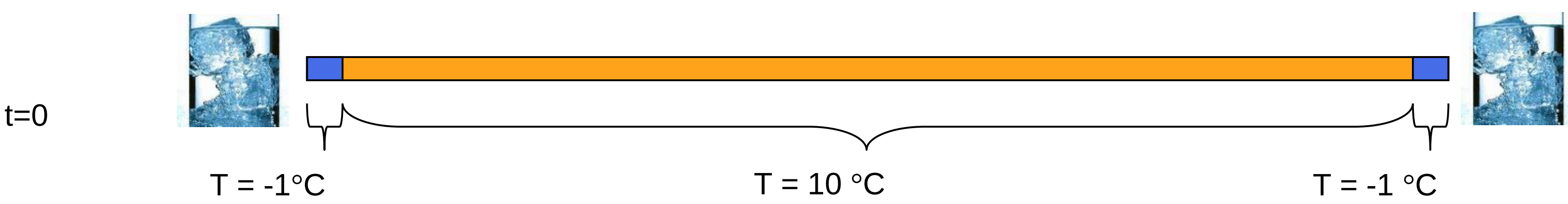
Exercise 18 – Heat Equation (2 / 10)

Example 1:

- Imagine a thin rod that is given an initial temperature distribution.



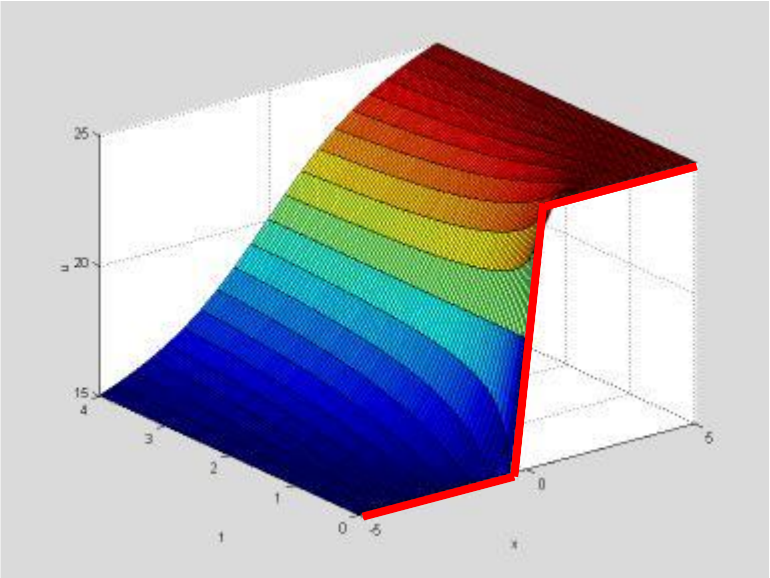
- Now the ends of the rod are kept at a fixed (and different) temperature; e.g., suppose at the start of the experiment ($t = 0$), both ends are immediately plunged into ice water or held against something cold (boundary conditions).

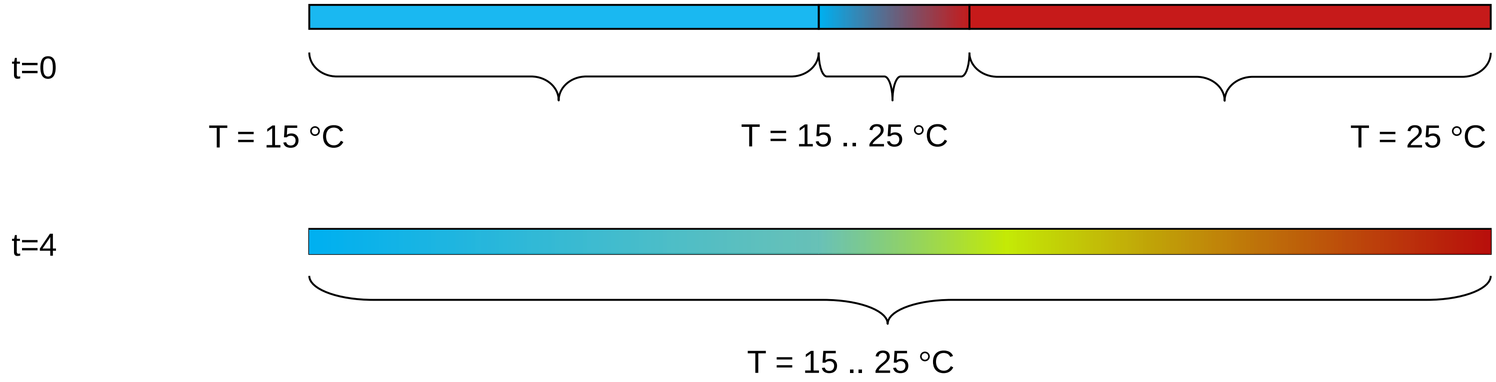


- We are interested in how the temperatures along the rod vary with time, i.e. we look for $u(x, t)$ for $t > 0$.

Example 2:

- $u(x,t)$ as 2D-function



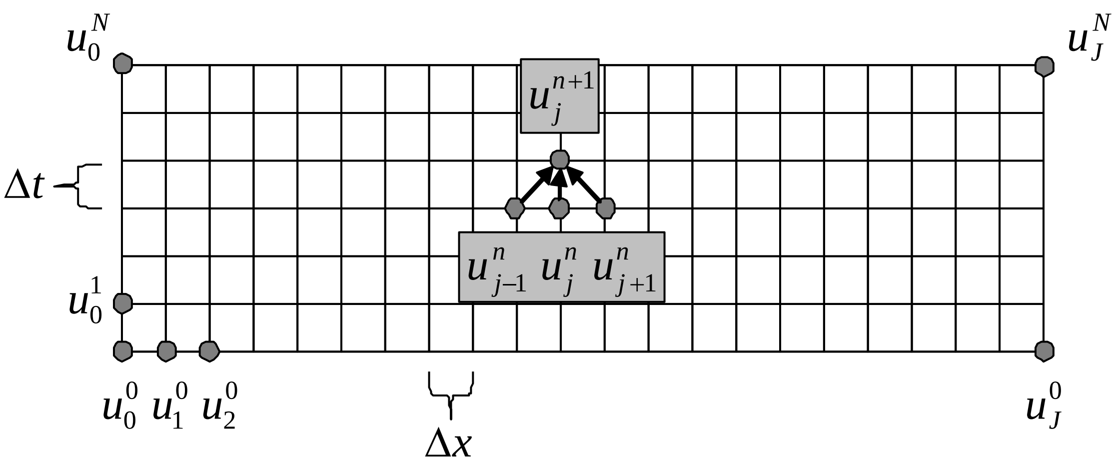


Numerical Solution:

- On the computer we can only keep track of the temperature *u* at a discrete set of times and a discrete set of positions …

$$u_j^n = u(j\Delta x, n\Delta t)$$

- … for *j* in *{0,…,J}* and *n* in *{0,…,N}* with some constants *N* and *J*.



Numerical Solution:

- Rewriting the partial differential equation in terms of *finite difference approximations* to the derivatives, we get

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = k\,\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta x^2}$$

- These are the simplest approximations. Thus if for a particular n, we know the values of for all j, we can solve the equation above to find for each *j* (see figure above):

$$u_j^{n+1} = u_j^n + k\,\frac{\Delta t}{\Delta x^2}\left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right)$$

- In other words, this equation tells us how to find the temperature distribution at time step $n + 1$ given the temperature distribution at time step n. At the endpoints $j = 0$ and $j = J$ we ignore the equation above and apply the boundary conditions instead.

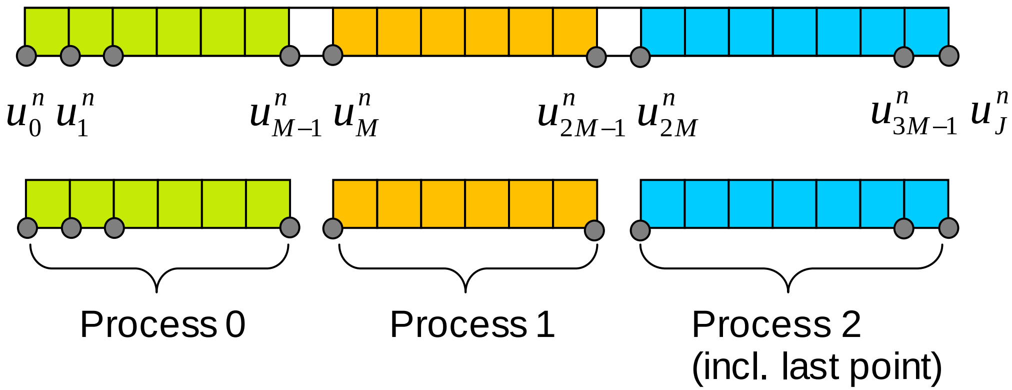
Exercise 18 – Heat Equation (6 / 10)

Code-Snippet:

Serial Code

```
const double TempLeft  = -1.0;
const double TempRight = -1.0;
const double TempMid   = 10.0;

const int N = 1000; // time steps
const int J =  100; // space discretization
[…]
double dx = 1.0/J;
double dt = 0.5*dx*dx;
[…]
double* uk0 = malloc((J+1)*sizeof(double)); // take right point into account
double* uk1 = malloc((J+1)*sizeof(double)); // we have u[0],u[1],...,u[J]
double* ukt; // swap fields

InitFields(uk0,uk1,J+1, 0,TempLeft, J,TempRight, TempMid);
AppendFile("data0", uk1,0.0, 0,J+1);

for (int n=1; n<=N; ++n)  // all time steps
{
  for (int j=1; j<J; ++j) // all locations, 1..J-1
  {
    uk1[j] = uk0[j] + dt/(dx*dx) * (uk0[j-1]-2*uk0[j]+uk0[j+1]);
  }

  ukt = uk0; uk0 = uk1; uk1 = ukt;
}
```

- **Parallelization:**
  - Obviously, the inner loop (j) can be easily parallelized to P processes (assume the number of cells is divisible without remainder). We set $M = J/P$.

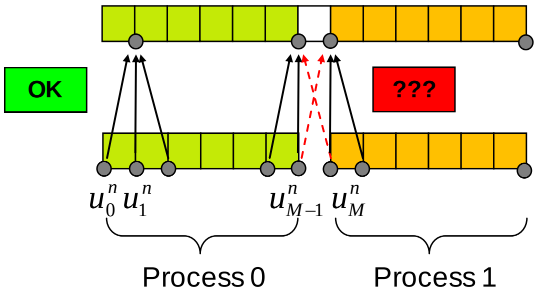


  - Example above: $J = 21, P = 3, M = 7$
  - Chosen that partition scheme every process can compute all inner points individually!

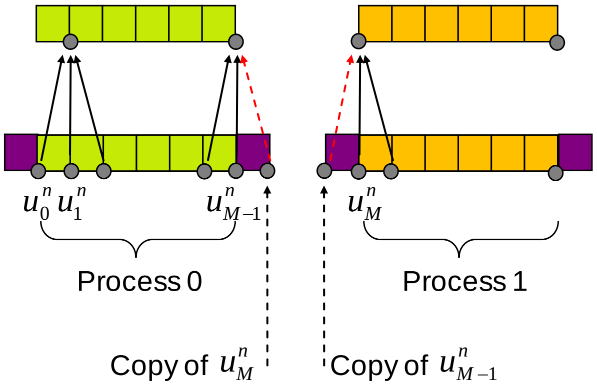
Exercise 18 – Heat Equation (8 / 10)

Parallelization:

- There is only one problem left, namely getting the endpoint-values, or local boundary points, of each partition.



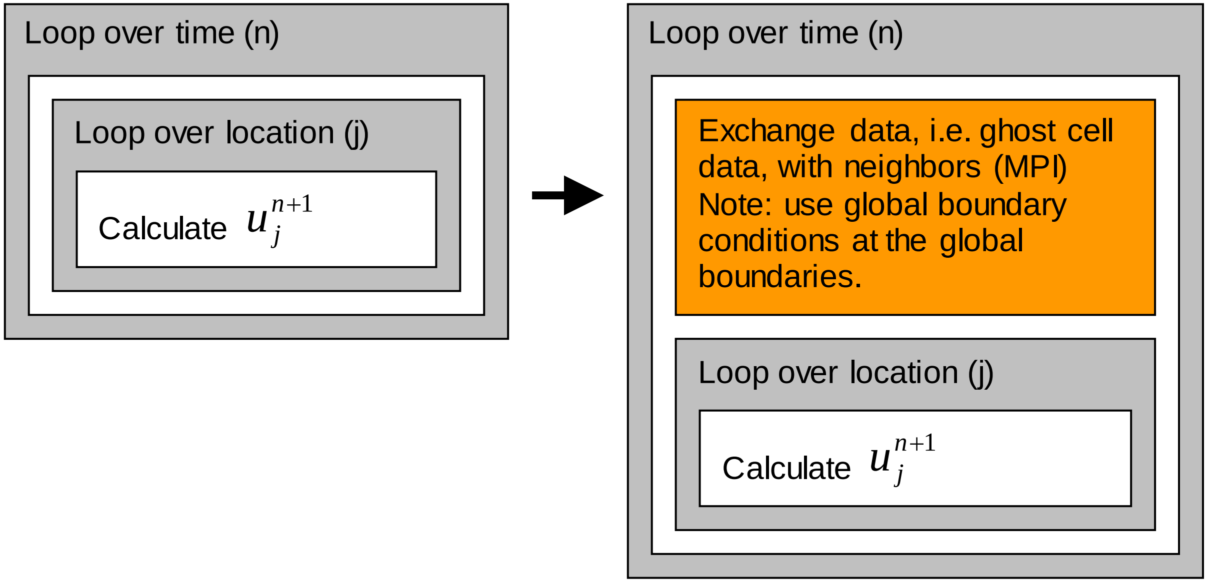
- One solution to this problem is to introduce additional cells, so-called *ghost cells*, being copies of the boundary cells of the neighbour processes, cf. the following figure.



Exercise 18 – Heat Equation (9 / 10)

Parallelization:

- Then we have to change the loops as follows:



Exercise 18 – Heat Equation (10 / 10)

- **Exercise:**
  - Extend the serial code such that the problem can be computed parallel.
  - Use non-blocking communication.