

Message Passing with MPI

PPCES 2018

Joachim Protze / Marc-André Hermanns
IT Center / JARA-HPC

Slides by Hristo Iliev

■ Motivation

■ Part 1

- Concepts
- Point-to-point communication
- Non-blocking operations

■ Part 2

- Collective operations
- Communicators
- User datatypes

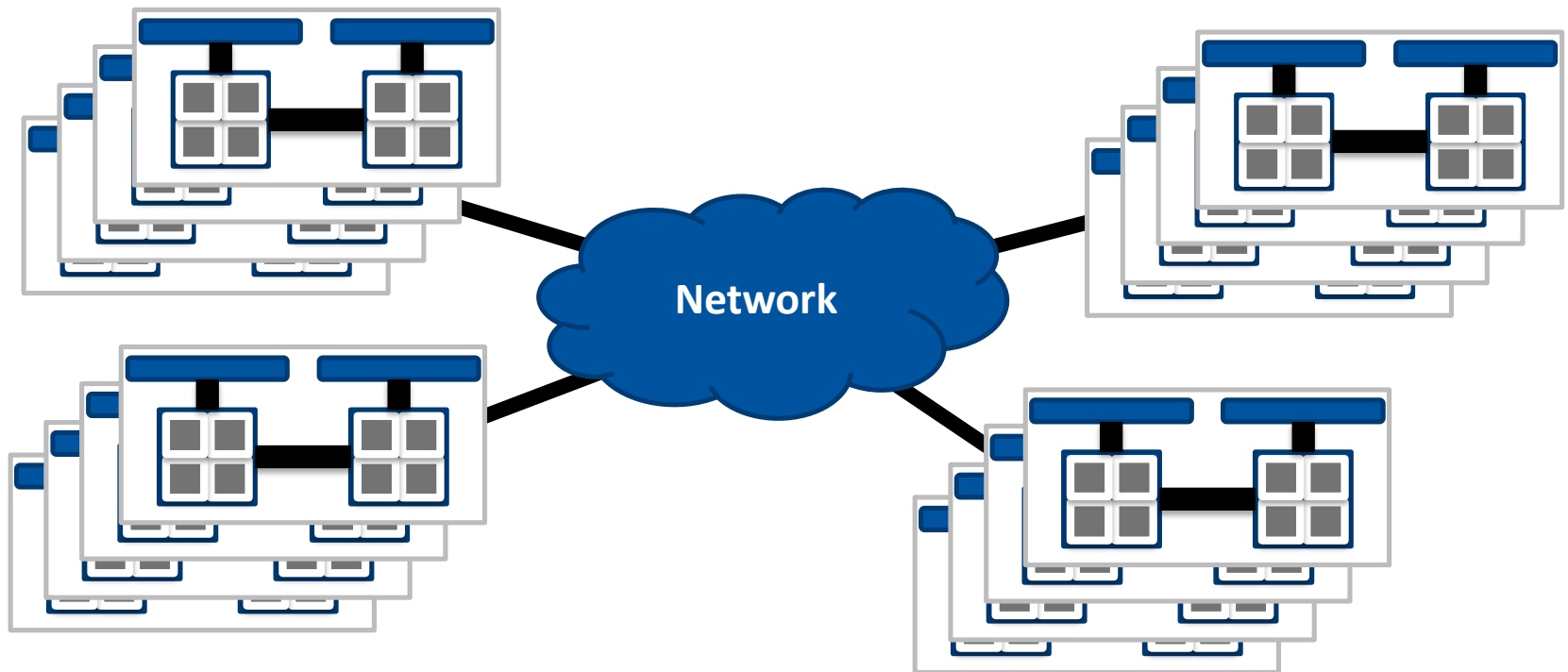
■ Part 3

- Hybrid parallelisation
- Common parallel patterns

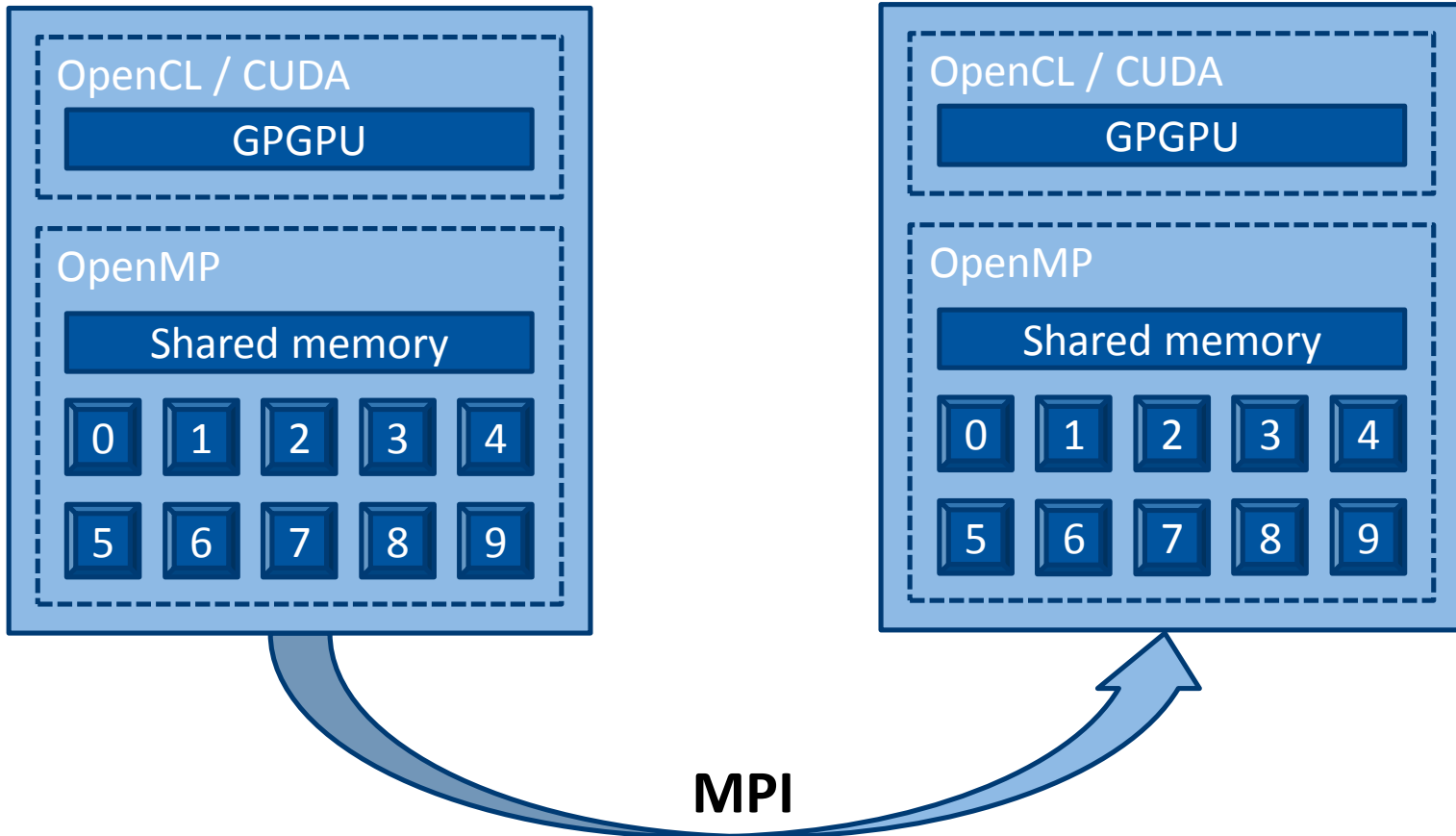
- **MPI is sufficiently abstract so it runs perfectly fine on a single node:**
 - it doesn't care where processes are located as long as they can communicate
 - message passing implemented using shared memory and IPC
 - all details hidden by the MPI implementation;
 - usually faster than sending messages over the network;
 - but...
- **... this is far from optimal:**
 - MPI processes are separate (heavyweight) OS processes
 - portable data sharing is hard to achieve
 - lots of program control / data structures have to be duplicated (uses memory)
 - reusing cached data is practically impossible

■ Clusters

- HPC market is at large dominated by distributed memory *multicomputers*: *clusters* and specialised *supercomputers*
- Nodes have no direct access to other nodes' memory and run a separate copy of the (possibly stripped down) OS



■ Hierarchical mixing of different programming paradigms



- Most MPI implementations are threaded (e.g. for non-blocking requests) but not thread-safe.
- Four levels of threading support in increasing order:

Level identifier	Description
<code>MPI_THREAD_SINGLE</code>	Only one thread may execute
<code>MPI_THREAD_FUNNELED</code>	Only the main thread may make MPI calls
<code>MPI_THREAD_SERIALIZED</code>	Only one thread may make MPI calls at a time
<code>MPI_THREAD_MULTIPLE</code>	Multiple threads may call MPI at once with no restrictions

- All implementations support `MPI_THREAD_SINGLE`, but some do not support `MPI_THREAD_MULTIPLE`.

■ Initialise MPI with thread support:

```
MPI_Init_thread (int *argc, char ***argv, int required, int *provided)
```

```
SUBROUTINE MPI_INIT_THREAD (required, provided, ierr)
```

→ **required** specifies what thread level support one requires from MPI

→ **provided** is set to the actual thread level support provided

→ could be lower or higher than the required level – always check!

→ **MPI_Init** – equivalent to **required = MPI_THREAD_SINGLE**

■ The thread that calls **MPI_Init_thread** becomes the main thread

■ The level of thread support cannot be changed later

■ Obtain the provided level of thread support:

```
MPI_Query_thread (int *provided)
```

- If MPI was initialised by **MPI_Init_thread**, then **provided** is set to the same value as the one returned by the initialisation call
- If MPI was initialised by **MPI_Init**, then **provided** is set to an implementation specific default value

■ Find out if running in the main thread:

```
MPI_Is_thread_main (int *flag)
```

- **flag** set to **true** if the current thread is the main thread

- **The most common approach to hybrid programming**
 - Coarse-grained parallelisation with MPI
 - Fine-grained loop or task parallelisation with OpenMP

- **Different MPI implementations provide varying degree of support for threaded programs**
 - `MPI_THREAD_MULTIPLE` rarely implemented completely for all transports
 - Performance decrease due to locking overhead

- **Safest and most portable approach: Call MPI from the main thread only (and outside any OpenMP parallel region)**

■ Simple: Iterative processing with MPI only

```
double data[], localData[];

for (int iter = 0; iter < maxIters; iter++) {

    MPI_Scatter(data, count, MPI_DOUBLE,
               localData, count, MPI_DOUBLE,
               0, MPI_COMM_WORLD);

    for (int i = 0; i < count; i++)
        localData[i] = exp(localData[i]);

    MPI_Gather(localData, count, MPI_DOUBLE,
              data, count, MPI_DOUBLE,
              0, MPI_COMM_WORLD);

}
```

- Safe: MPI called outside any OpenMP parallel region

```
double data[], localData[];

for (int iter = 0; iter < maxIters; iter++) {

    MPI_Scatter(data, count, MPI_DOUBLE,
               localData, count, MPI_DOUBLE,
               0, MPI_COMM_WORLD);

    #pragma omp parallel for
    for (int i = 0; i < count; i++)
        localData[i] = exp(localData[i]);

    MPI_Gather(localData, count, MPI_DOUBLE,
              data, count, MPI_DOUBLE,
              0, MPI_COMM_WORLD);

}
```

■ Advanced: MPI called by the master OpenMP thread only

```
double data[], localData[];
#pragma omp parallel
for (int iter = 0; iter < maxIters; iter++) {
    #pragma omp master
    MPI_Scatter(data, count, MPI_DOUBLE,
               localData, count, MPI_DOUBLE,
               0, MPI_COMM_WORLD);

    #pragma omp barrier
    #pragma omp for
    for (int i = 0; i < count; i++)
        localData[i] = exp(localData[i]);
    #pragma omp master
    MPI_Gather(localData, count, MPI_DOUBLE,
              data, count, MPI_DOUBLE,
              0, MPI_COMM_WORLD);

    #pragma omp barrier
}
```

■ Adventurous: MPI called by a single OpenMP thread at a time

```
MPI_Init_thread(NULL, NULL, MPI_THREAD_SERIALIZED, &provided);

double data[], localData[];
#pragma omp parallel
for (int iter = 0; iter < maxIters; iter++) {
    #pragma omp single
    MPI_Scatter(data, count, MPI_DOUBLE,
               localData, count, MPI_DOUBLE,
               0, MPI_COMM_WORLD);

    #pragma omp for
    for (int i = 0; i < count; i++)
        localData[i] = exp(localData[i]);
    #pragma omp single
    MPI_Gather(localData, count, MPI_DOUBLE,
              data, count, MPI_DOUBLE,
              0, MPI_COMM_WORLD);
}
```

■ MPI was not designed initially with multithreading in mind

- Single rank (end-point) per process per communicator
- Addressing individual threads is tricky (and mostly hacky)

■ MPI and OpenMP IDs live in orthogonal spaces

- MPI rank $\in [0, \text{\#procs}-1]$ **MPI_Comm_rank()**
- OpenMP thread ID $\in [0, \text{\#threads}-1]$ **omp_get_thread_num()**
- Hybrid rank:thread $\in [0, \text{\#procs}-1] \times [0, \text{\#threads}-1]$

Field	Value source	Remark
source rank	Sender process rank	Automatically copied, no control over it
destination rank	user-supplied	Only one rank per process
tag	user-supplied	Free to choose
communicator	user-supplied	Multiple communicators possible

■ Tags as thread IDs

→ Each MPI message carries a tag with at least 15 bits of user-supplied data

■ Simple idea: use tag value to address individual threads

→ (+) straightforward to implement

→ (+) very large number of threads per process addressable

→ (-) not possible to further differentiate the messages

→ (-) no information about the sending thread retained

■ Tags as thread IDs

→ Each MPI message carries a tag with at least 15 bits of user-supplied data

■ Better idea: multiplex destination thread ID with tag value

→ e.g. 7 bits for tag value (0..127) and 8 bits for thread ID (0..255)

→ (+) still possible to differentiate the messages

→ (-) wildcard receives not trivial to implement

→ (-) no information about the sending thread retained

■ Tags as thread IDs

→ Each MPI message carries a tag with at least 15 bits of user-supplied data

■ Even better idea: multiplex source and destination thread IDs with tag value

→ suitable for MPI implementations that allow more than 15 bits for tag value

→ Open MPI and Intel MPI both allow tag values from 0 to $2^{31}-1$

→ (+) still possible to differentiate the messages

→ (+) information about the sending thread retained

→ (-) wildcard receives not trivial to implement

→ (-) not portable to MPI implementations with smaller tag space

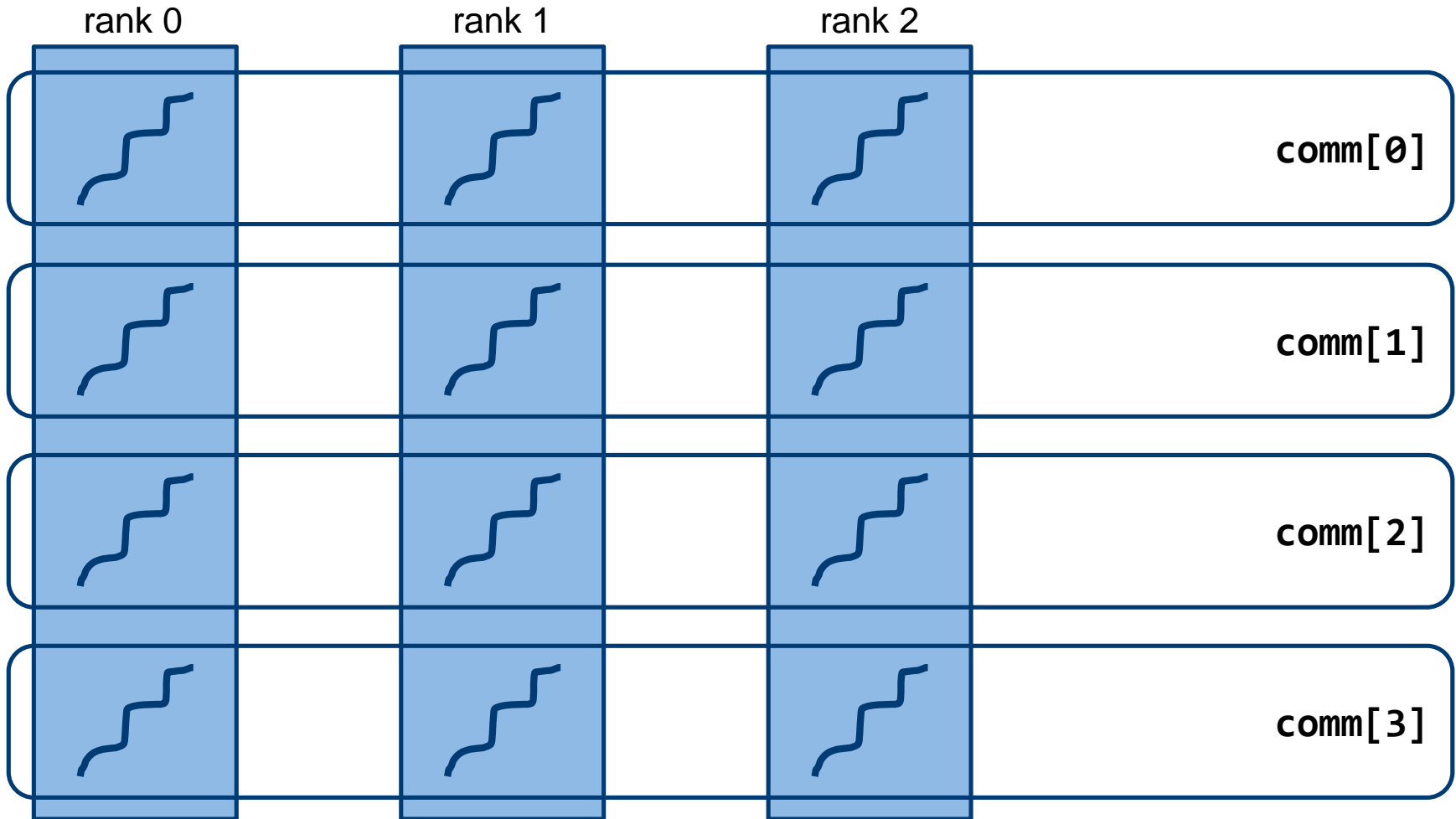
■ Multiplex source and destination thread IDs with tag value

```
#define MAKE_TAG (tag, stid, dtid) \  
    (((tag) << 16) | ((stid) << 8) | (dtid))  
  
// Send data to drank:dtid with tag mytag  
  
MPI_Send(data, count, MPI_FLOAT, drank,  
    MAKE_TAG(mytag, omp_get_thread_num(), dtid),  
    MPI_COMM_WORLD);  
  
-----  
  
// Receive data from srank:stid with a specific tag mytag  
  
MPI_Recv(data, count, MPI_FLOAT, srank,  
    MAKE_TAG(mytag, stid, omp_get_thread_num()),  
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

■ Multiplex source and destination thread IDs with tag value

```
#define GET_TAG(val) \  
    ((val) >> 16)  
#define GET_SRC_TID(val) \  
    (((val) >> 8) & 0xff)  
#define GET_DST_TID(val) \  
    ((val) & 0xff)  
  
// Wildcard receive from srank:stid with any tag  
  
MPI_Probe(srank, MPI_ANY_TAG, MPI_COMM_WORLD, &status);  
if (GET_SRC_TID(status.MPI_TAG) == stid &&  
    GET_DST_TID(status.MPI_TAG) == omp_get_thread_num())  
{  
    MPI_Recv(data, count, MPI_FLOAT, srank, status.MPI_TAG,  
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
}
```

- An alternative is the use of multiple communicators



■ Multiple communicators

```
MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);
MPI_Comm comm[nthreads], tcomm;
#pragma omp parallel private(tcomm) num_threads(nthreads)
{
    MPI_Comm_dup(MPI_COMM_WORLD, &comm[omp_get_thread_num()]);
    tcomm = comm[omp_get_thread_num()];
    -----
    // Sender
    MPI_Send(data, count, MPI_FLOAT, omp_get_thread_num(),
             drank, comms[dtid]);
    -----
    // Receiver
    MPI_Recv(data, count, MPI_FLOAT, stid, srank, tcomm,
             &status);
    -----
    MPI_Comm_free(&comm[omp_get_thread_num()]);
}
```

■ Both official MPI libraries support threads:

requested	Open MPI	Open MPI mt	Intel MPI
SINGLE	SINGLE	SINGLE	SINGLE
FUNNELED	FUNNELED	FUNNELED	FUNNELED
SERIALIZED	SERIALIZED	SERIALIZED	SERIALIZED
MULTIPLE	SERIALIZED	MULTIPLE	MULTIPLE

■ Open MPI

→ Switch to an MT version, e.g. `module switch openmpi openmpi/1.10.6mt`

→ **OMPI 1.x does not support MPI_THREAD_MULTIPLE over InfiniBand**

■ Intel MPI

→ Compile with `-openmp`, `-parallel` or `-mt_mpi`

■ Sample hybrid job for Open MPI

```
#!/usr/bin/env zsh
# 16 MPI procs x 6 threads = 96 cores
#BSUB -x
#BSUB -a openmpi
#BSUB -n 16
# 12 cores/node / 6 threads = 2 processes per node
#BSUB -R "span[ptile=2]"

module switch openmpi openmpi/1.10.6mt
# 6 threads per process
# Pass OMP_NUM_THREADS on to all MPI processes
$MPIEXEC $FLAGS_MPI_BATCH -x OMP_NUM_THREADS=6 \
    program.exe <args>
```

■ Sample hybrid job for Intel MPI

```
#!/usr/bin/env zsh
# 16 MPI procs x 6 threads = 96 cores
#BSUB -x
#BSUB -a intelmpi
#BSUB -n 16
# 12 cores/node / 6 threads = 2 processes per node
#BSUB -R "span[ptile=2]"

module switch openmpi intelmpi
# 6 threads per process
# Pass OMP_NUM_THREADS on to all MPI processes
$MPIEXEC $FLAGS_MPI_BATCH -genv OMP_NUM_THREADS 6 \
    program.exe <args>
```


- **Beware of possible data races:**

- messages, matched by **MPI_Probe** in one thread, can be received by a matching receive in another thread, stealing the message from the first one

- Problem solved in MPI-3 with **MPI_Mprobe** (see MPI documentation)

- **MPI provides no way to address specific threads in a process**

- clever use of message tags

- clever use of many communicators

- MPI-4 will (hopefully) provide a better solution – MPI Endpoints

- **In general, thread-safe MPI implementations perform worse than non-thread-safe because of the added synchronisation overhead**

- **Don't use Open MPI on our cluster if full thread support is required!**



■ Motivation

■ Part 1

- Concepts
- Point-to-point communication
- Non-blocking operations

■ Part 2

- Collective operations
- Communicators
- User datatypes

■ Part 3

- Hybrid parallelisation
- Common parallel patterns

■ The run time of a program consists of two parts:

- sequential (non-parallelisable) part: T_s
- parallelisable part: T_p
- total execution time: $T = T_s + T_p$
- serial share: $s = T_s / T$



■ An n-fold parallelisation yields:

- total execution time: $T_n = T_s + T_p / n$
- parallel speedup: $S_n = T / T_n = n / [1 + (n-1).s]$
- parallel efficiency: $E_n = T / (n.T_n) = 1 / [1 + (n-1).s]$



■ Asymptotic values in the limit of infinite number of processors:

→ total execution time:

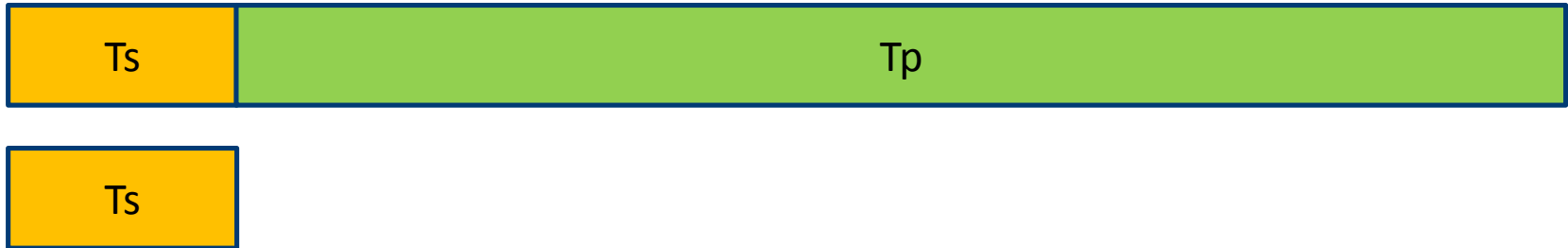
$$T_{\infty} = T_s + T_p / \infty = T_s$$

→ parallel speedup:

$$S_{\infty} = T / T_{\infty} = T / T_s = 1 / s$$

→ parallel efficiency:

$$E_{\infty} = 1 \text{ if } s = 0; E_{\infty} = 0 \text{ otherwise}$$



■ No parallel program can outrun the sum of its sequential parts

Keep this in mind for the rest
of your (parallel) life!

■ Parallelisation usually (if not always) introduces overhead:

→ communication is inherently serial → s increases

→ usually $E_n < 1$ – you use more CPU time than if you run the serial program

→ but sometimes cache effects result in $E_n > 1$ – superlinear speedup

■ Communication overhead increases with the number of processes:

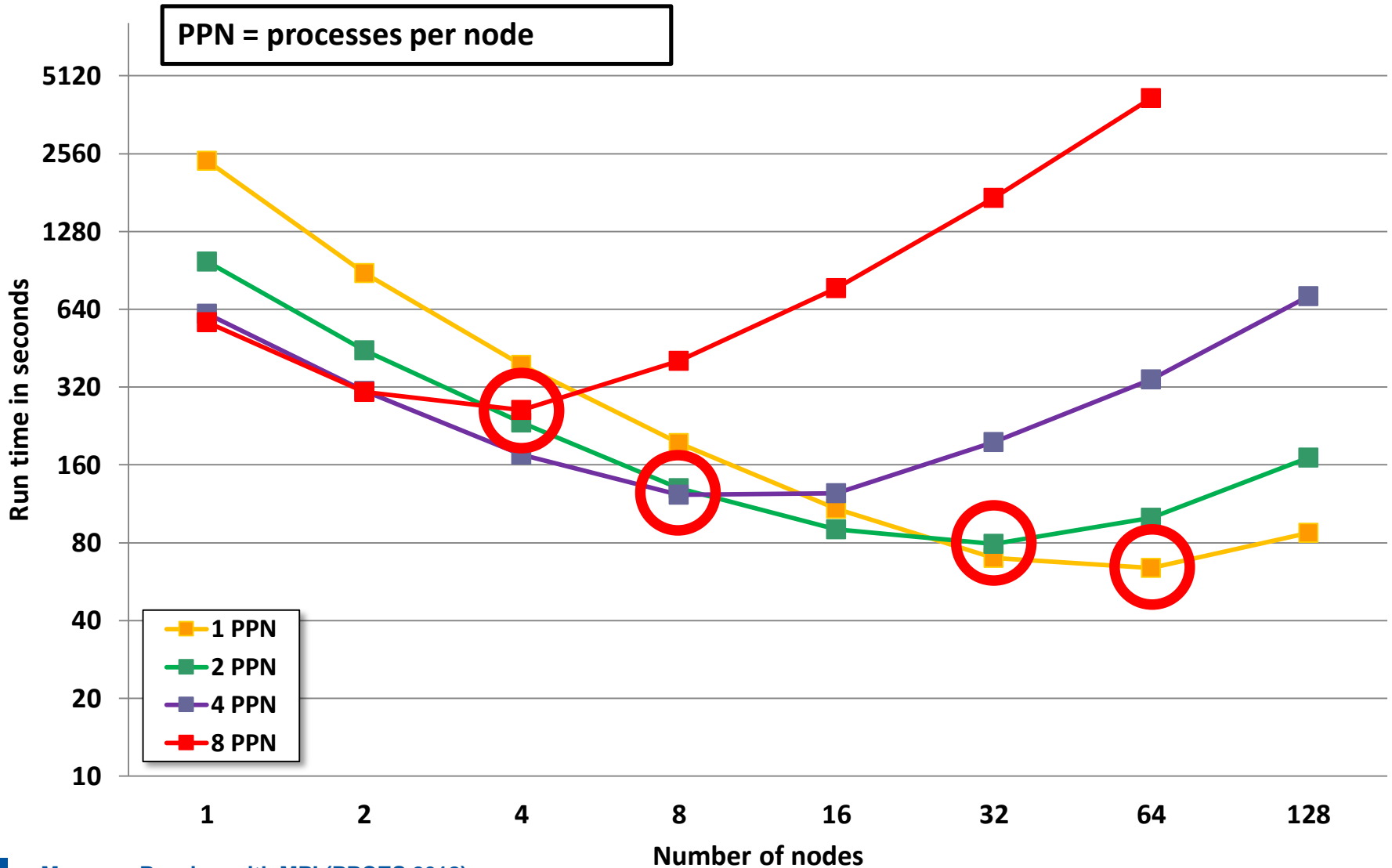
→ more processes → more messages (depends on the communication pattern)
especially true for the collective operations
(*you didn't re-implement them, did you?*)

→ more messages → more network latency → more serial time

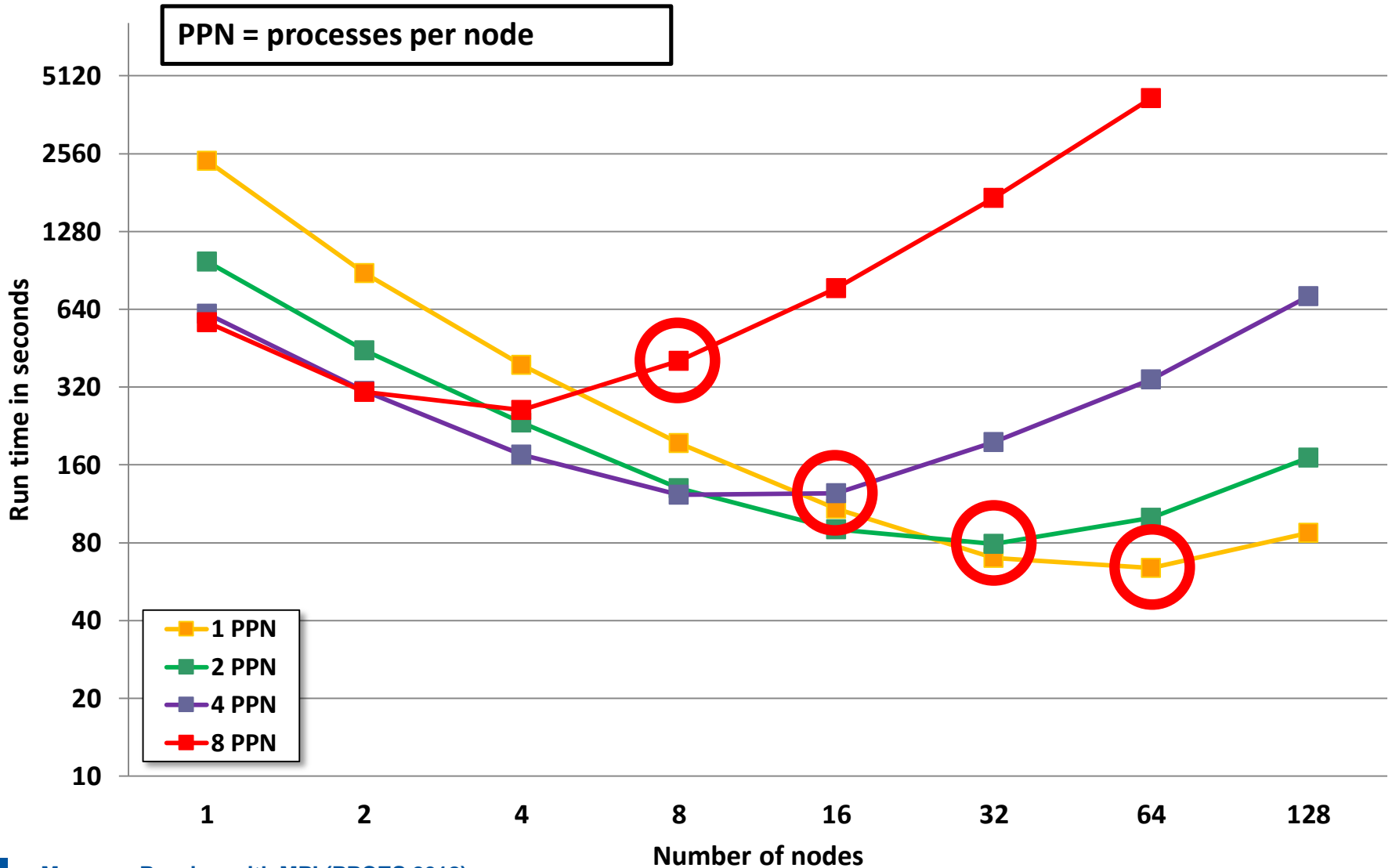
→ more serial time → lower parallel efficiency

→ with large process counts the overhead could negate the parallelisation gain

Amdahl's Law – The Ugly Truth™

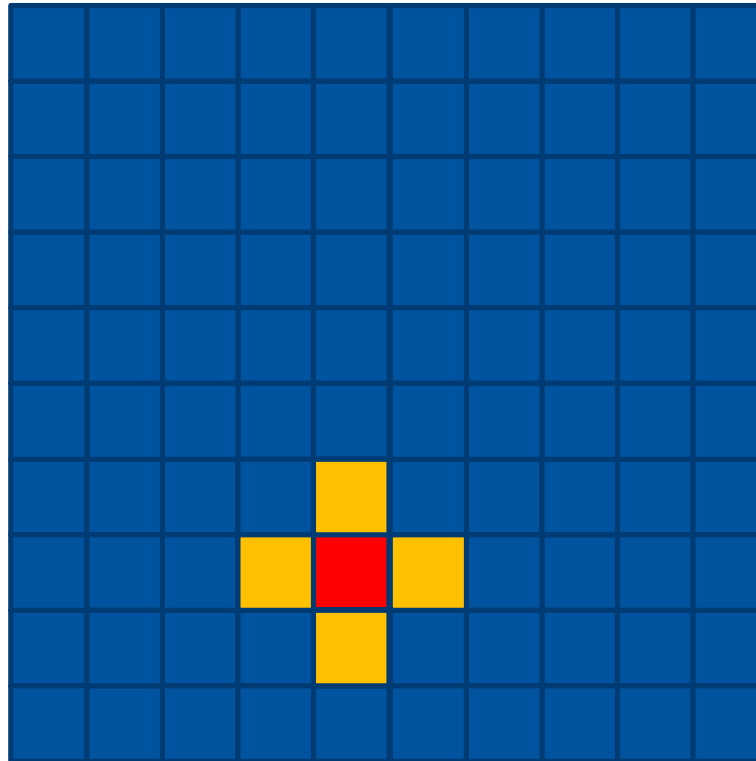


Amdahl's Law – The Ugly Truth™

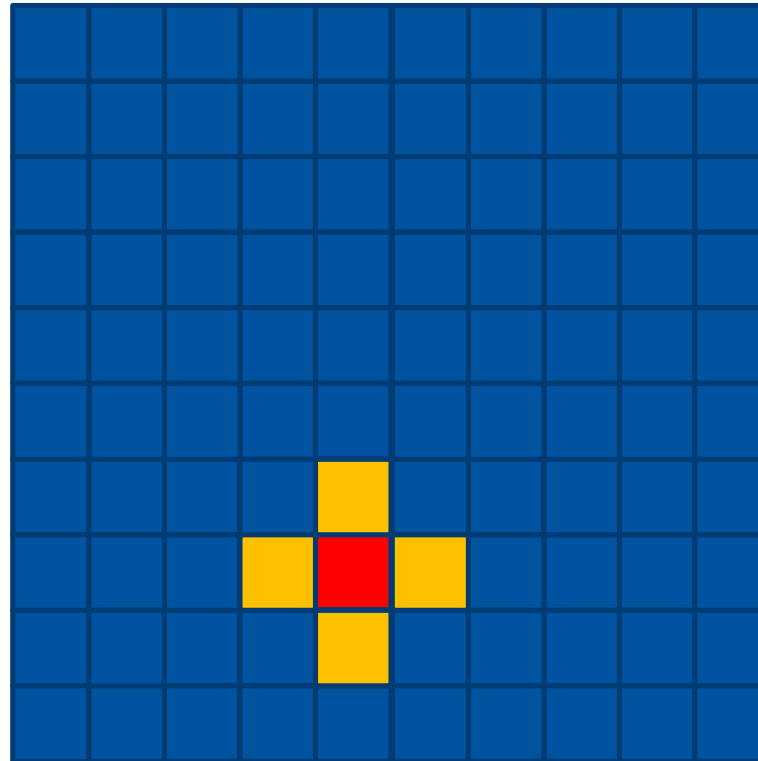


- When decomposing a problem often interdependent data ends up in separate processes.
- Example: iterative matrix update in a PDE solver:

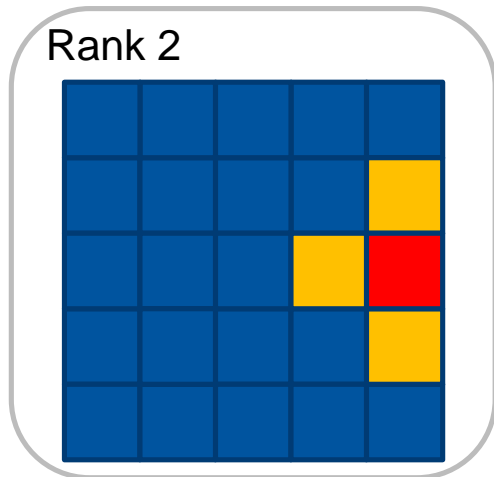
$$cell_{i,j} = f(cell_{i,j}; cell_{i-1,j}, cell_{i+1,j}, cell_{i,j-1}, cell_{i,j+1})$$



- **Domain decomposition strategy:**
 - Partition the domain into parts.
 - Each process works on one part only.



■ Domain decomposition



■ Communication

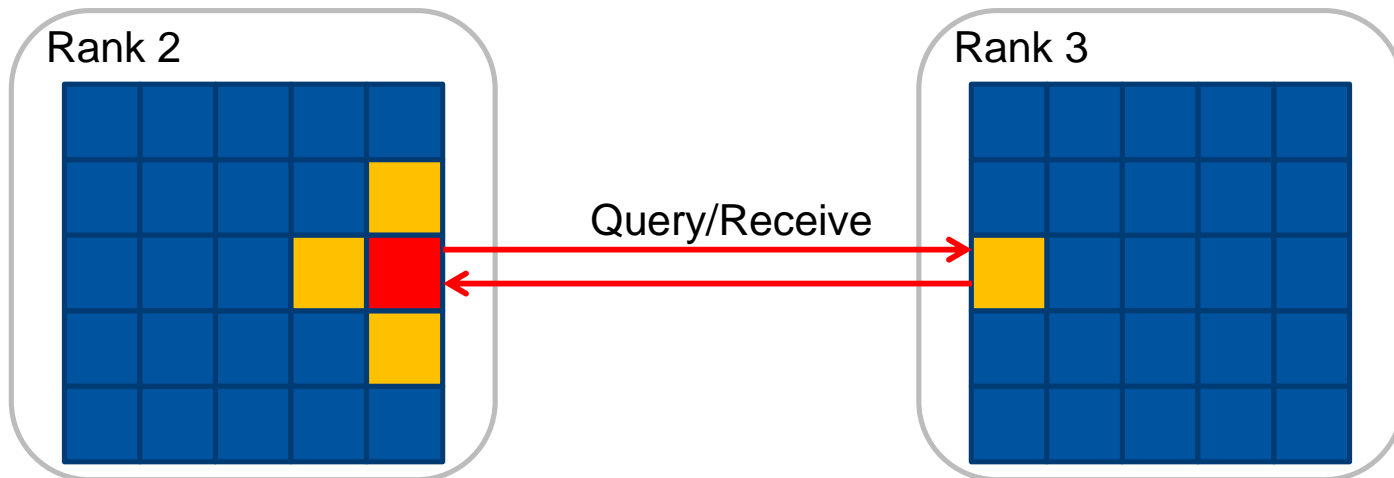
→ For every border cell communication with a neighbour rank is needed

■ Problems:

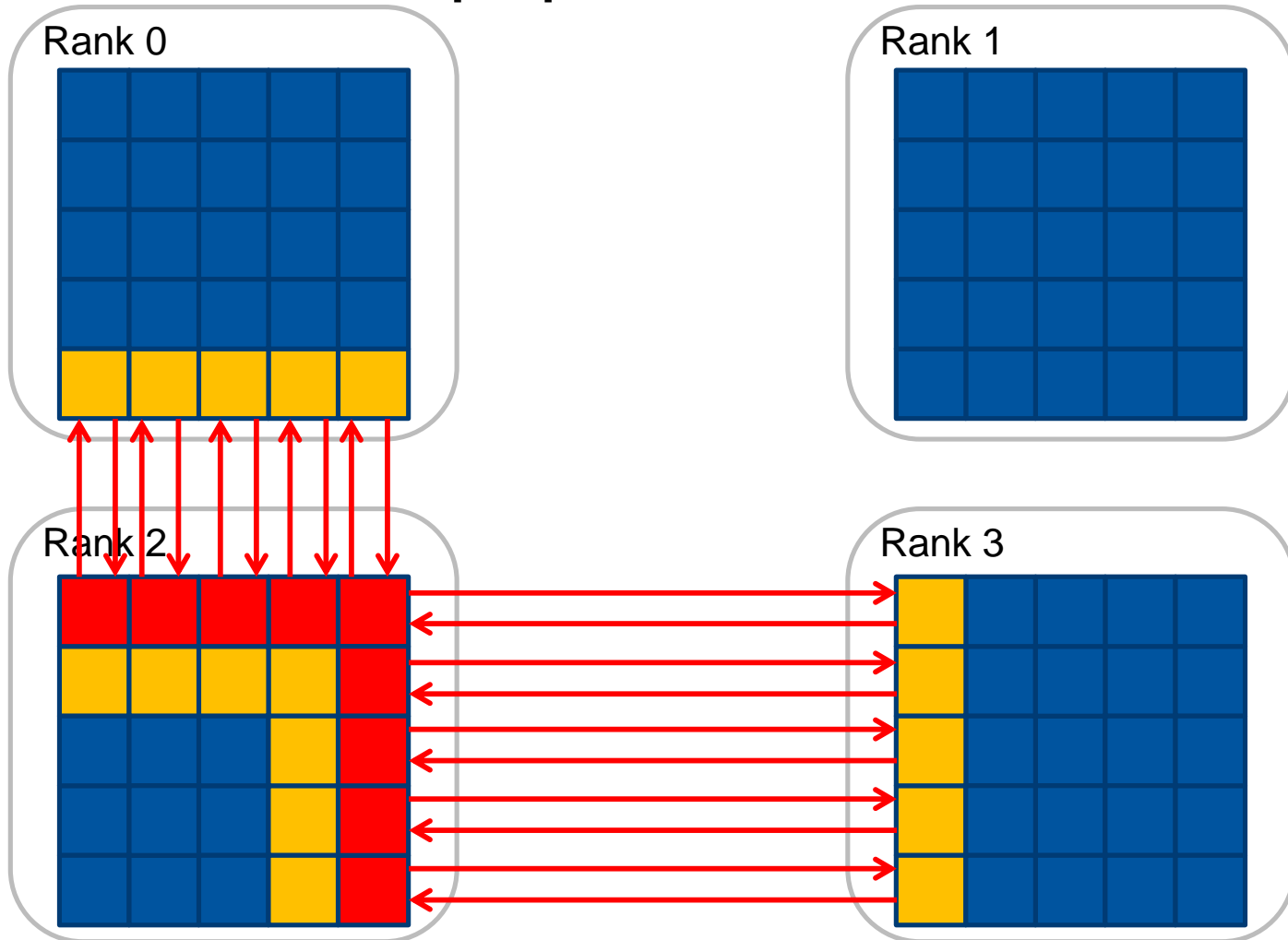
→ Introduces synchronisation on a very fine level

→ Lots of communication calls – highly inefficient

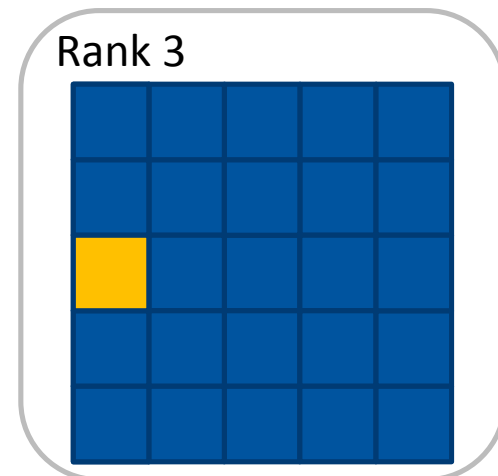
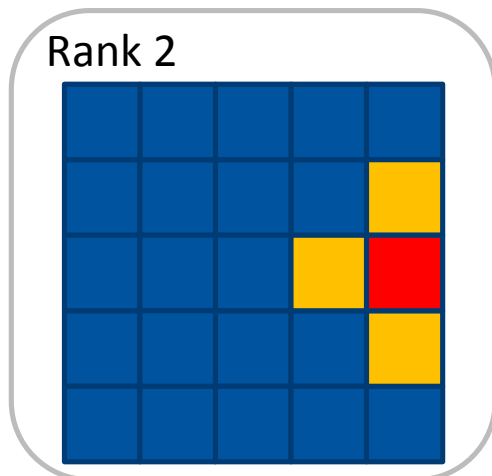
→ Performance can (and will) drastically suffer



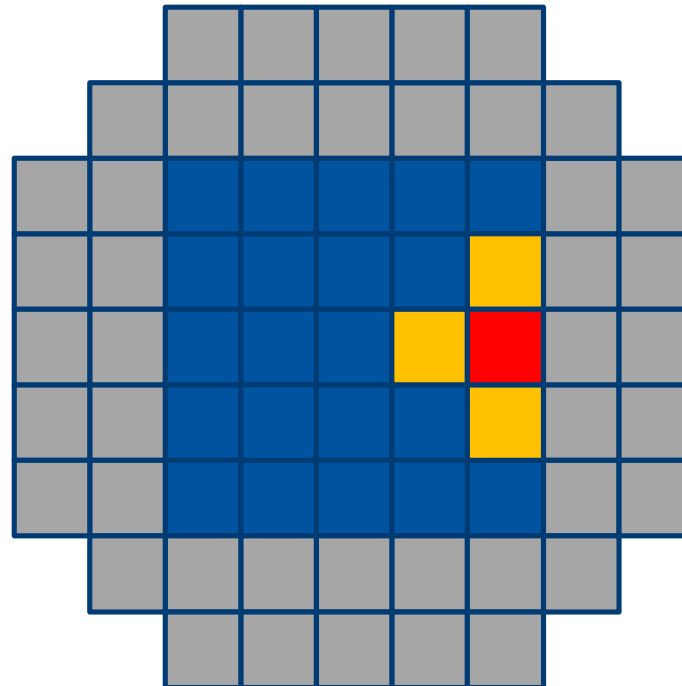
■ 10 communication calls per process



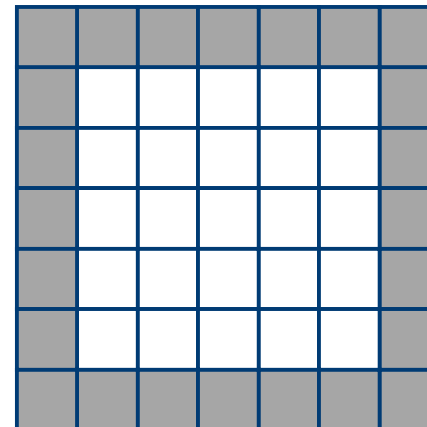
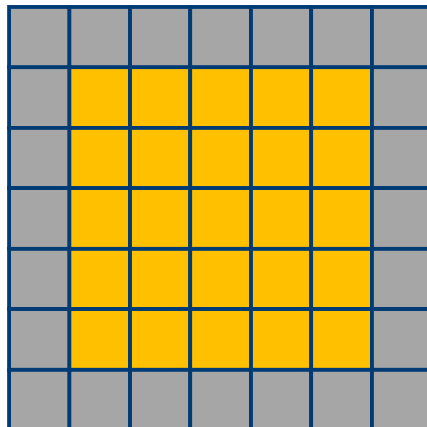
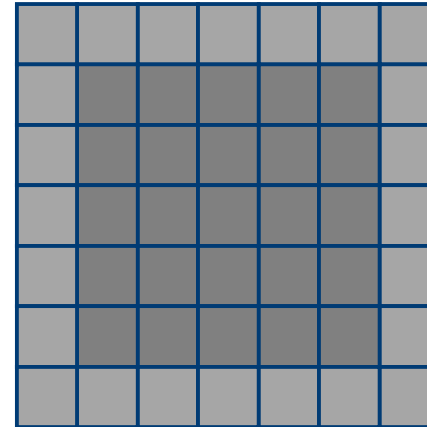
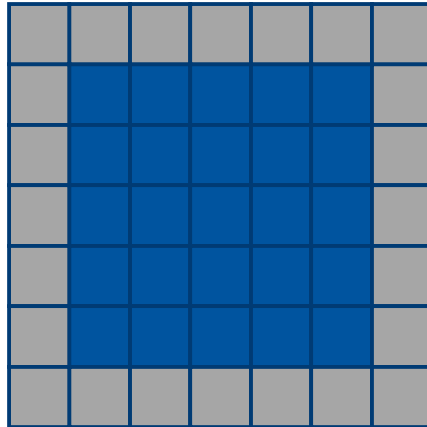
- Copy all the necessary data at the beginning of each iteration



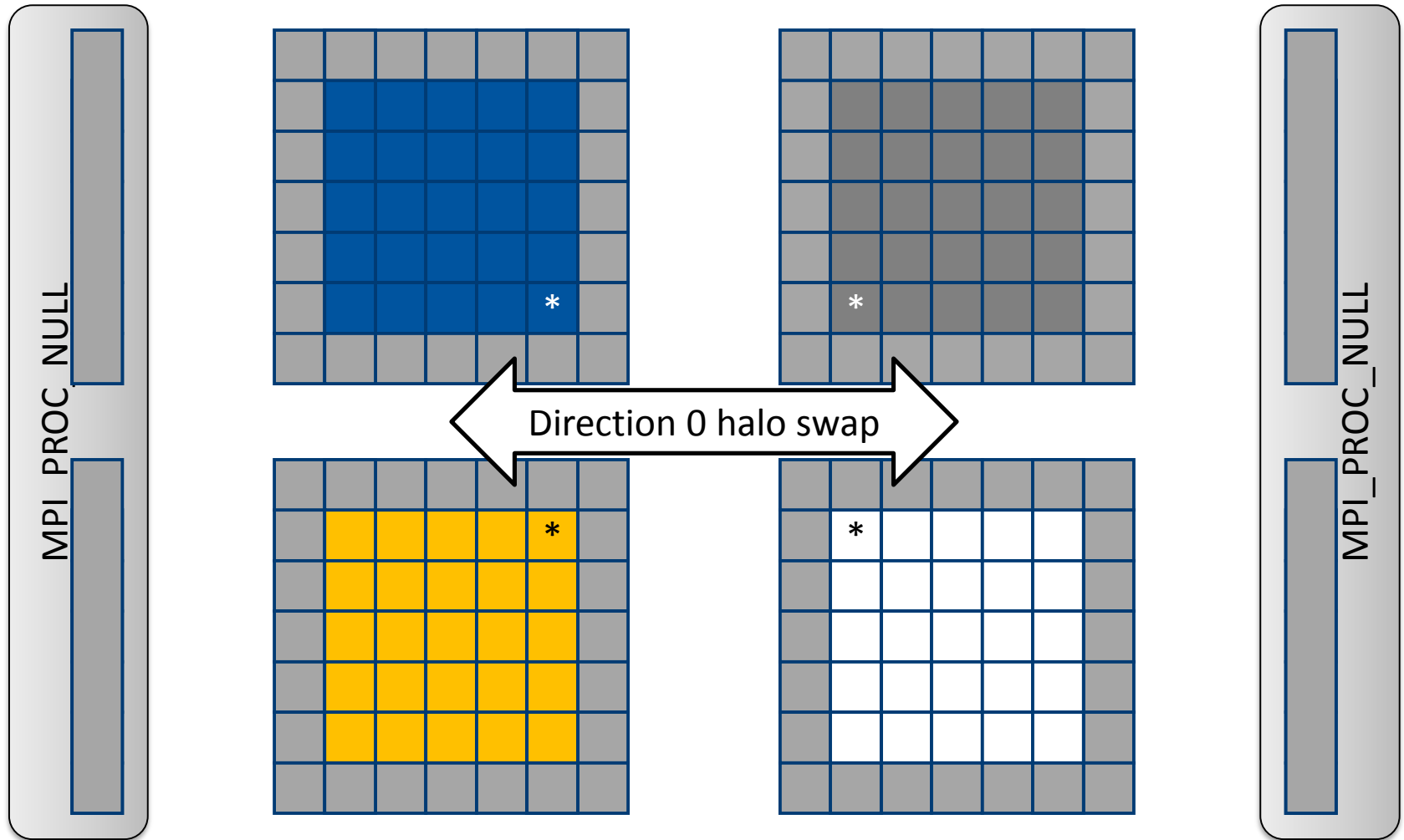
- Copies around the host domain are called *halos* or *ghost cells*
- Multi-level halos are also possible:
 - if required by the stencil
 - to reduce the number of communications



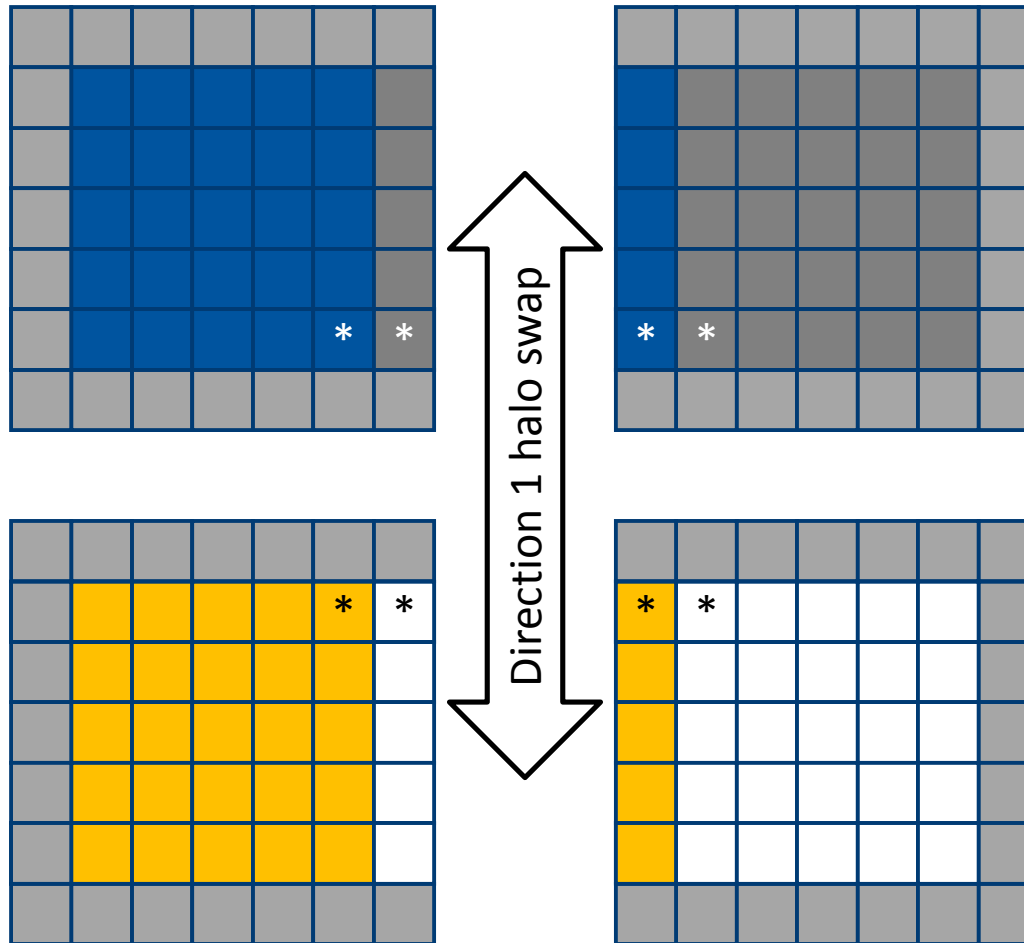
■ Halo Swap



■ Halo Swap



■ Halo Swap



■ Sample implementation

```
// Create a Cartesian topology
int dims[2] = {2,2}, periods[2] = {0,0};
MPI_Comm cart;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &cart);
// Ranks of the neighbours
int up, down, left, right;
MPI_Cart_shift(cart, 0, 1, &down, &up);
MPI_Cart_shift(cart, 1, 1, &left, &right);

// Halo swap in direction 0
MPI_Sendrecv(top_data, 1, row, up, 0,
             bottom_halo, 1, row, down, 0, cart);
MPI_Sendrecv(bottom_data, 1, row, down, 0,
             top_halo, 1, row, up, 0, cart);
// Halo swap in direction 1
MPI_Sendrecv(right_data, 1, column, right, 0,
             left_halo, 1, column, left, 0, cart);
MPI_Sendrecv(left_data, 1, column, left, 0,
             right_halo, 1, column, right, 0, cart);
```

- **Halo Swaps are locally synchronous, but combined they make a globally synchronous operation:**
 - initial process synchronisation is critical for the performance
 - one late process delays all the other processes
 - sensitivity to OS jitter (random delays introduced by OS or other processes)

- **Pros:**
 - idea comes naturally
 - simple to implement (two MPI send-receive calls per direction)

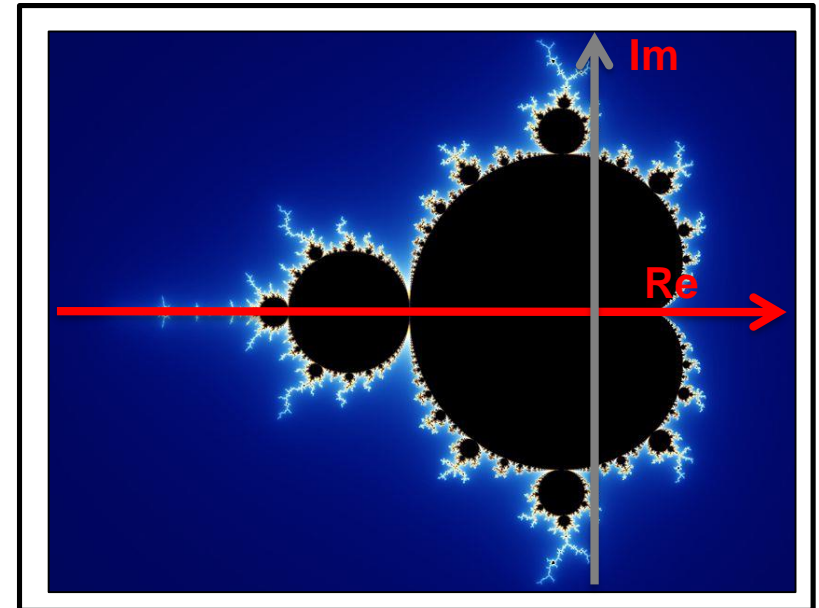
- **Cons:**
 - not suitable for problems where load imbalance may occur

■ Escape time colouring algorithm for the Mandelbrot set

```
For each image pixel (x, y):  
// (x, y) - scaled pixel coords  
c = x + iy  
z = 0  
iteration = 0  
maxIters = 1000  
while (|z|^2 <= 2^2 &&  
       iteration < maxIters)  
{  
    z = z^2 + c  
    iteration = iteration + 1  
}  
if (iteration == maxIters)  
    color = black  
else  
    color = iteration  
plot(x, y, color)
```

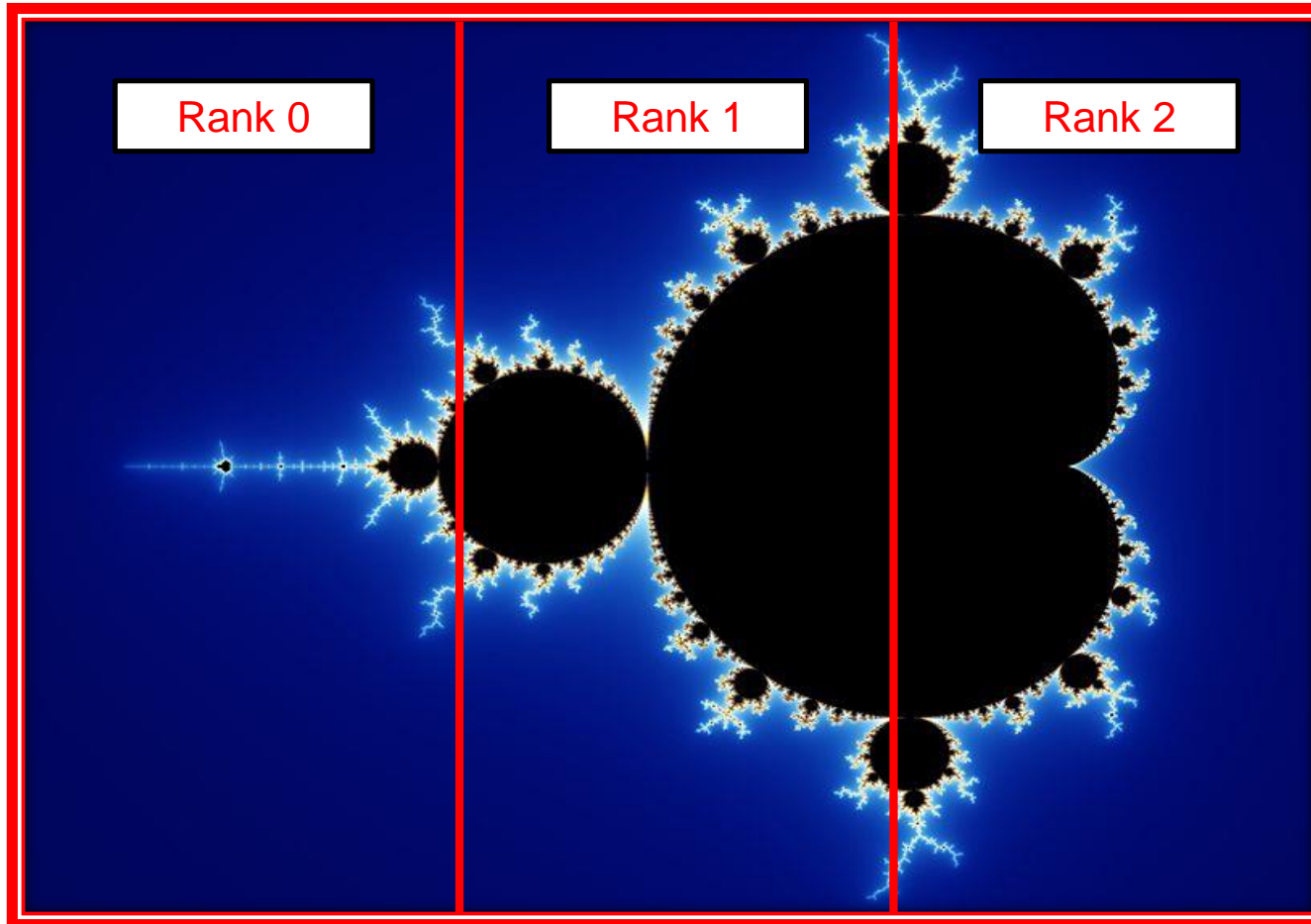
Does the complex series

$z_0 = 0; z_{n+1} = z_n^2 + c \quad z, c \in \mathbb{C}$
remain bounded?



■ Static work distribution:

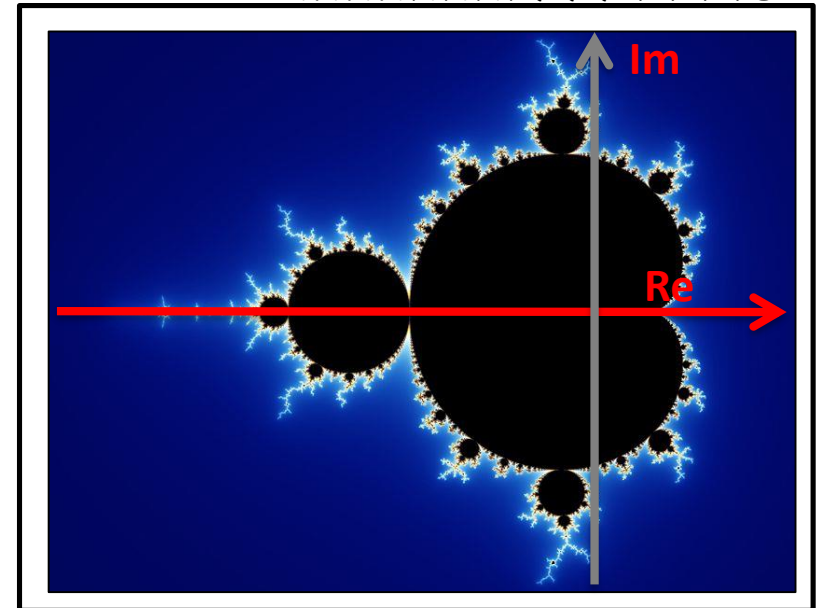
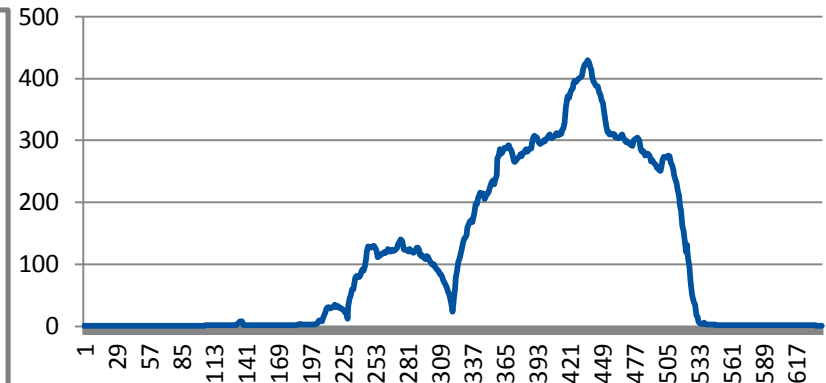
→ Every MPI rank works on 1/Nth of the problem (N – number of MPI ranks)



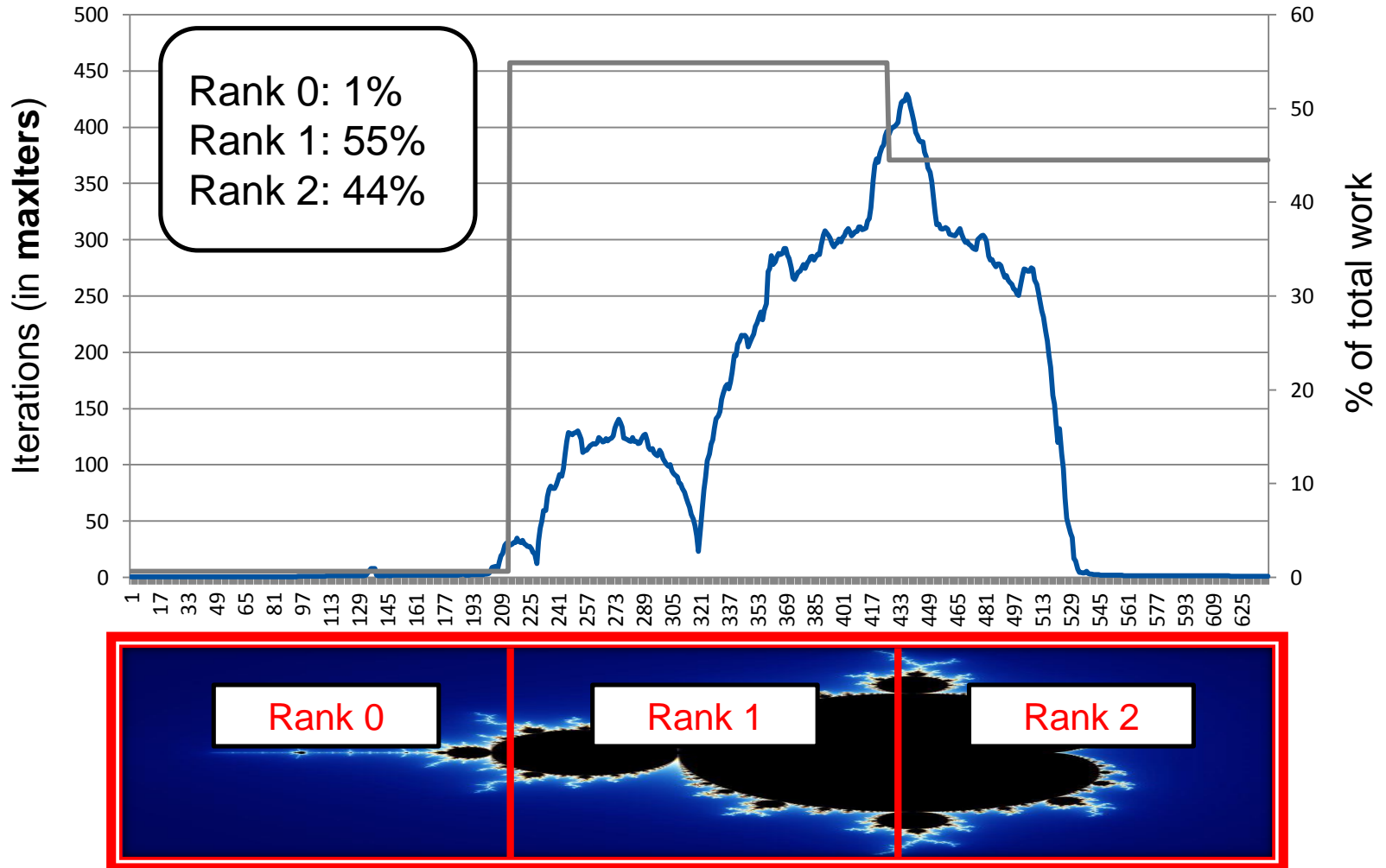
■ Mandelbrot set

```
For each image pixel (x, y):  
// (x, y) - scaled pixel coords  
c = x + iy  
z = 0  
iteration = 0  
maxIters = 1000  
while (|z|^2 <= 2^2 &&  
       iteration < maxIters)  
{  
    z = z^2 + c  
    iteration = iteration + 1  
}  
if (iteration == maxIters)  
    color = black  
else  
    color = iteration  
plot(x, y, color)
```

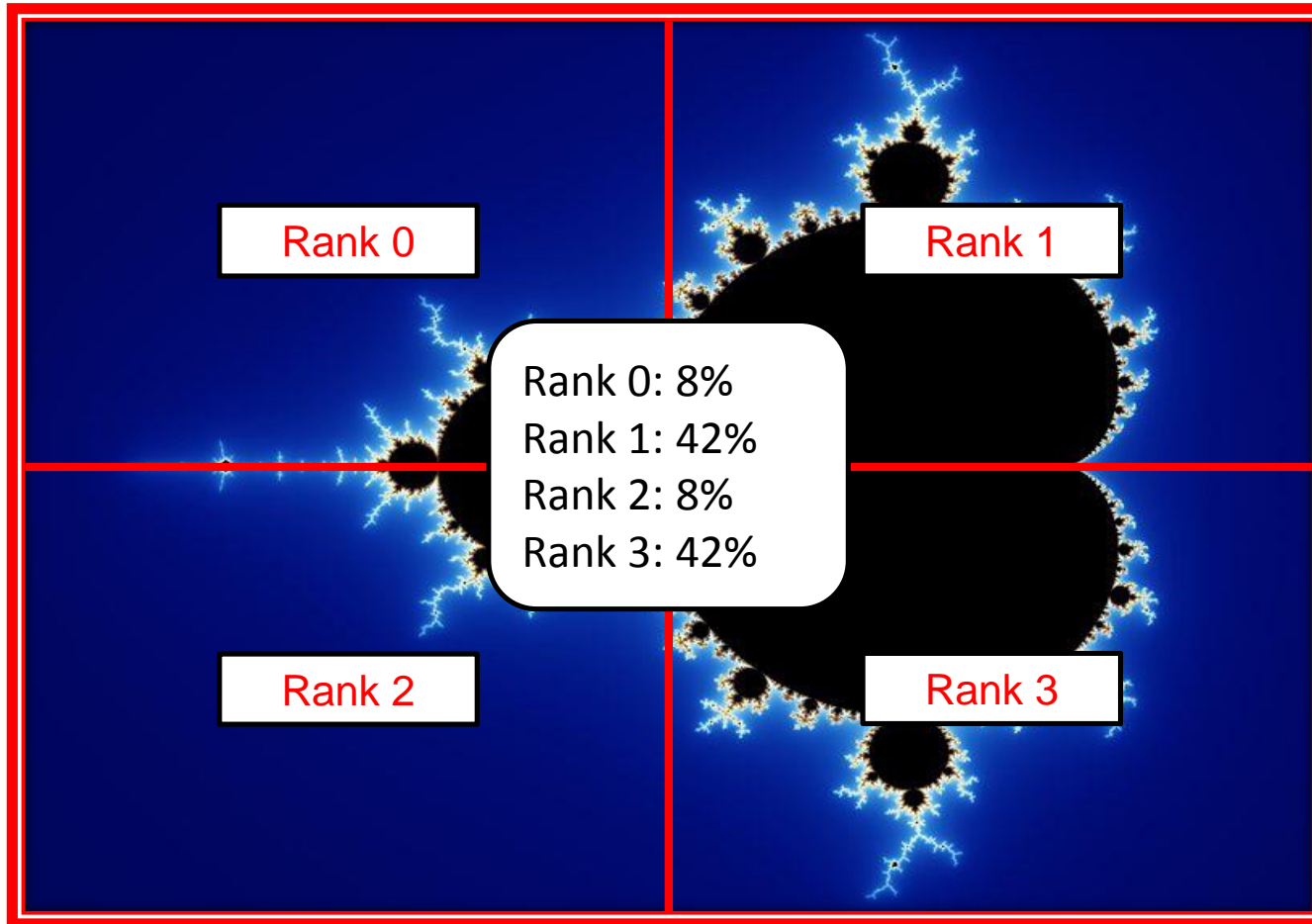
Iterations per image column (in maxIters)



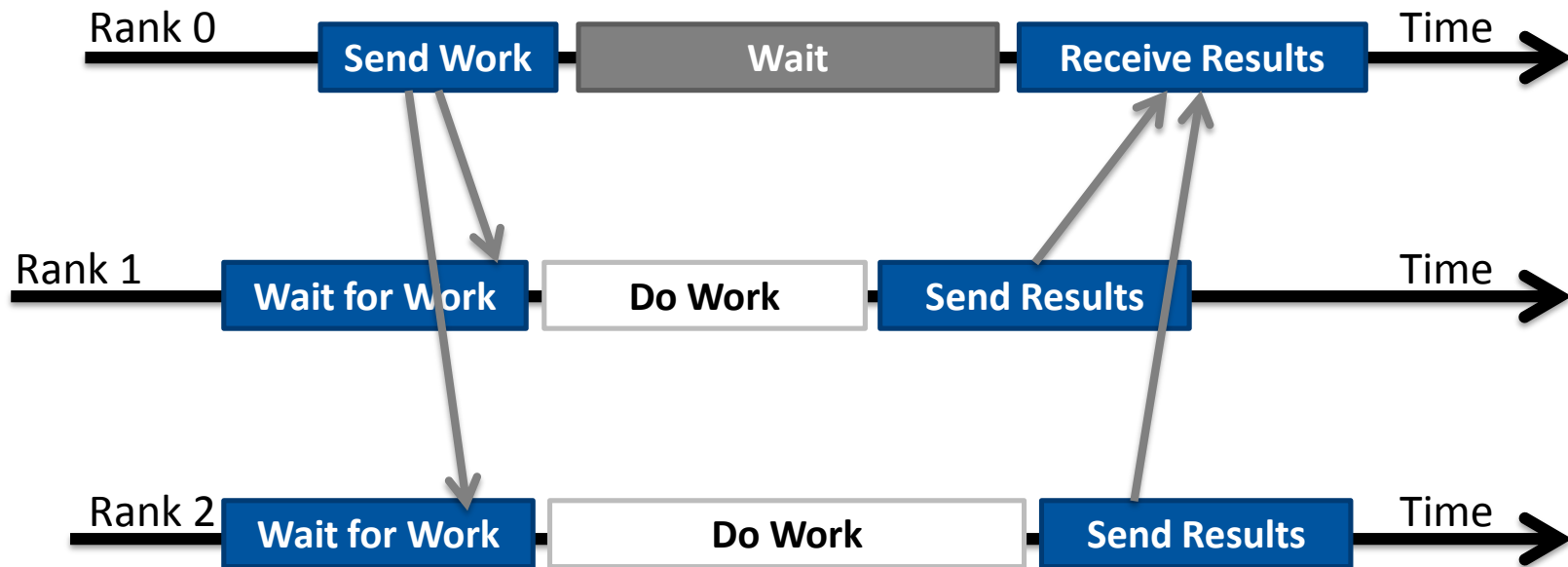
Computation complexity mapped to ranks



■ May be a different decomposition?



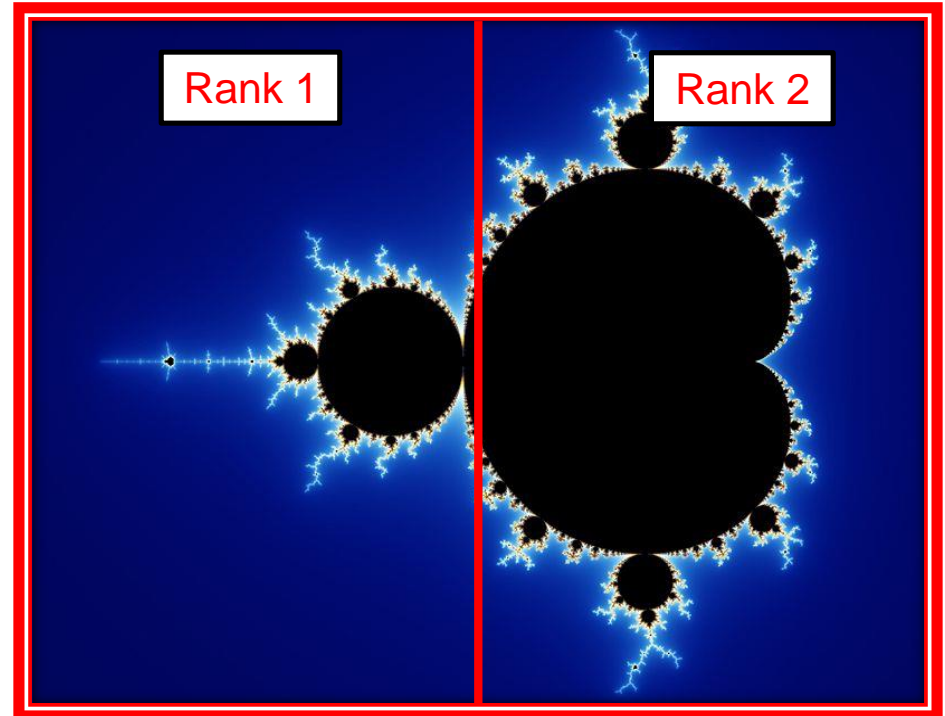
- **One process (controller) manages the work**
 - Work is split into many relatively small work items
- **Many other processes (workers) compute over the work items:**



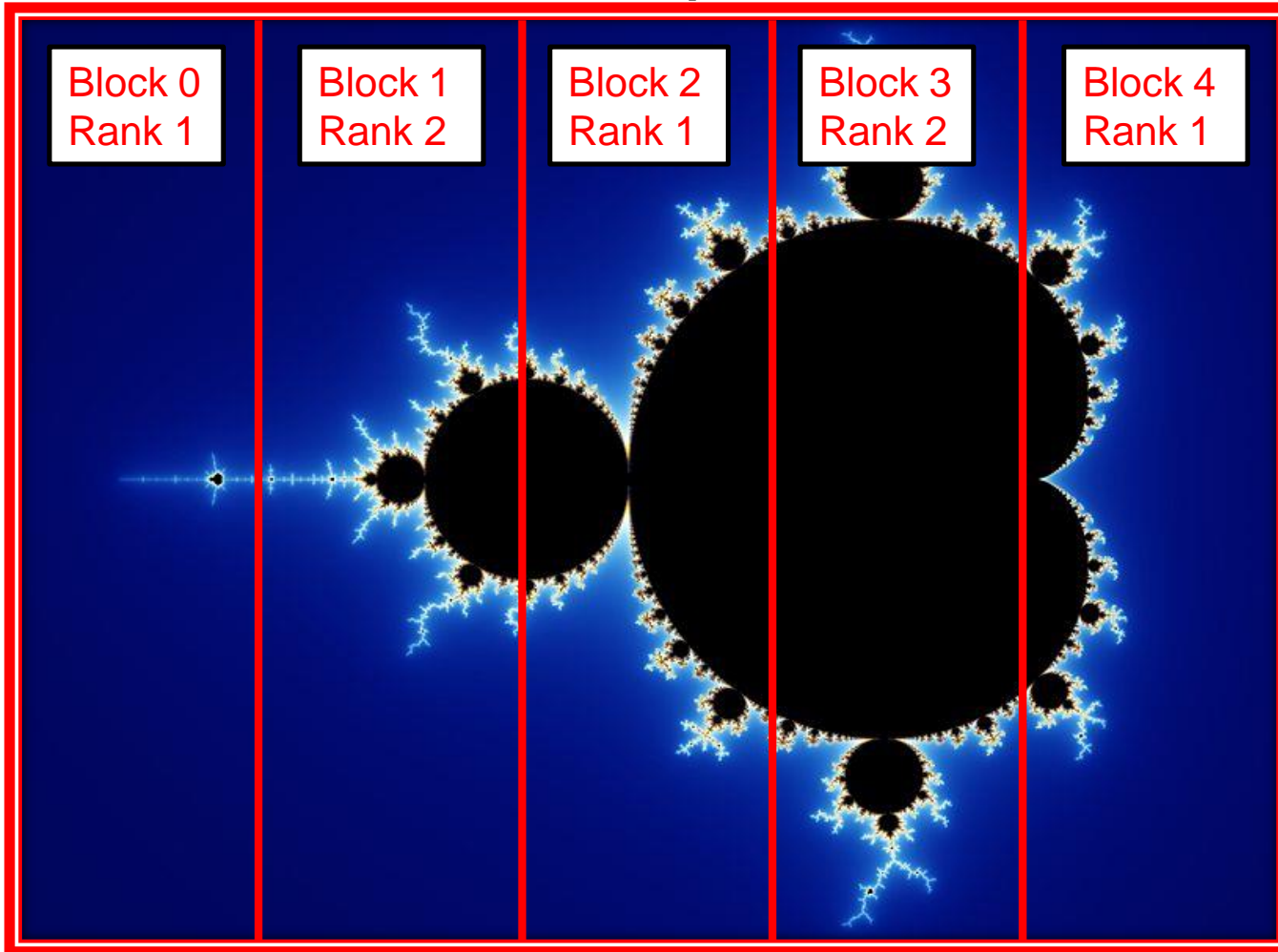
- **Above steps are repeated until all work items are processed**
- **Sometimes called “bag of jobs” pattern**

■ The algorithm:

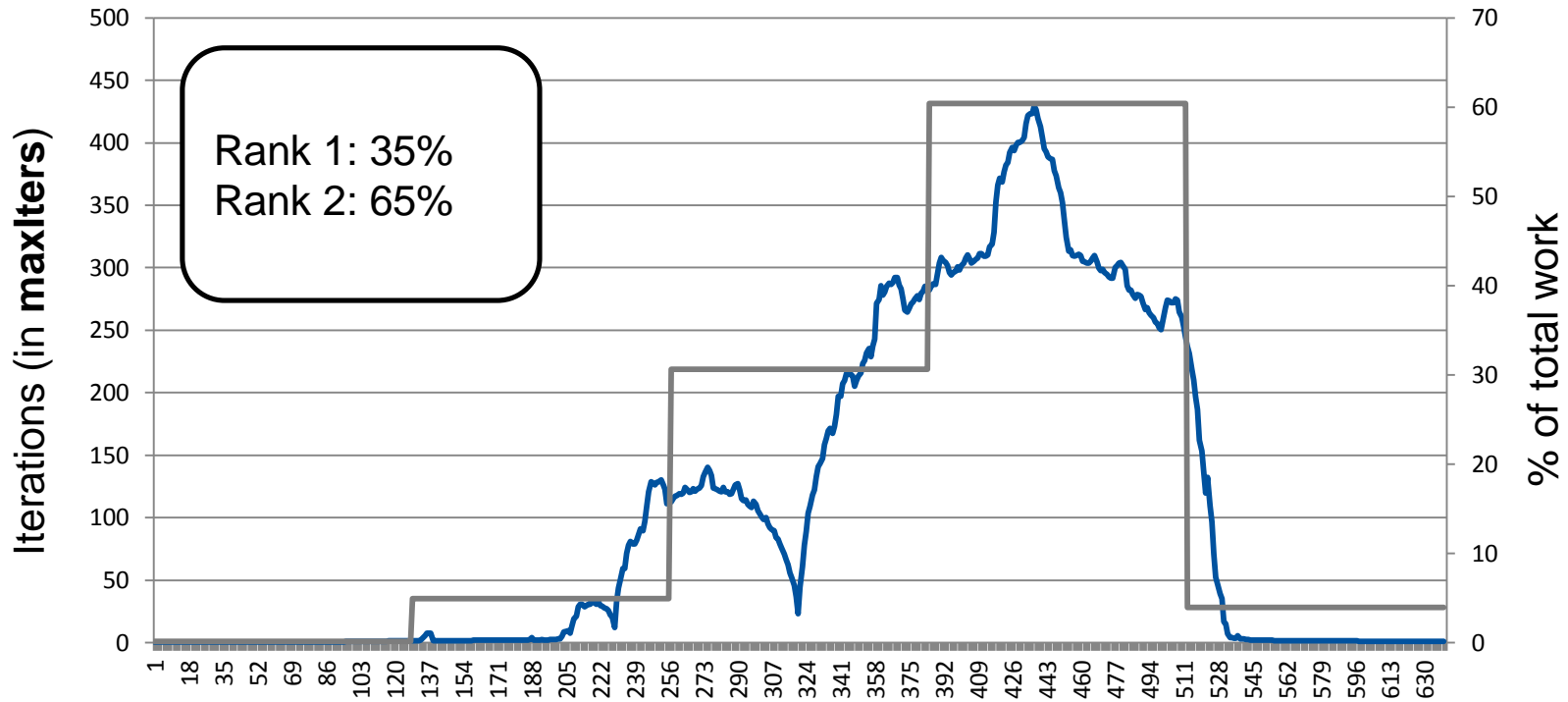
```
START
if (rank == 0)
{
    splitDomain;
    sendWorkItems;
    receiveItemResults;
    assembleResult;
    output;
}
else
{
    receiveWorkItems;
    processWorkItems;
    sendItemResults;
}
DONE
```



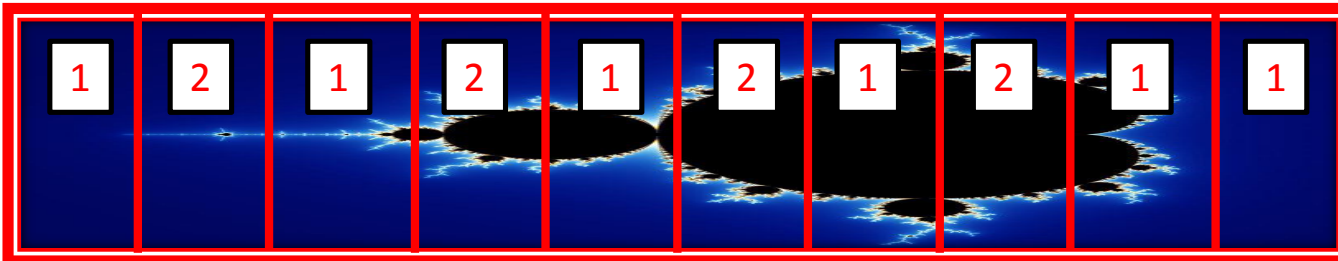
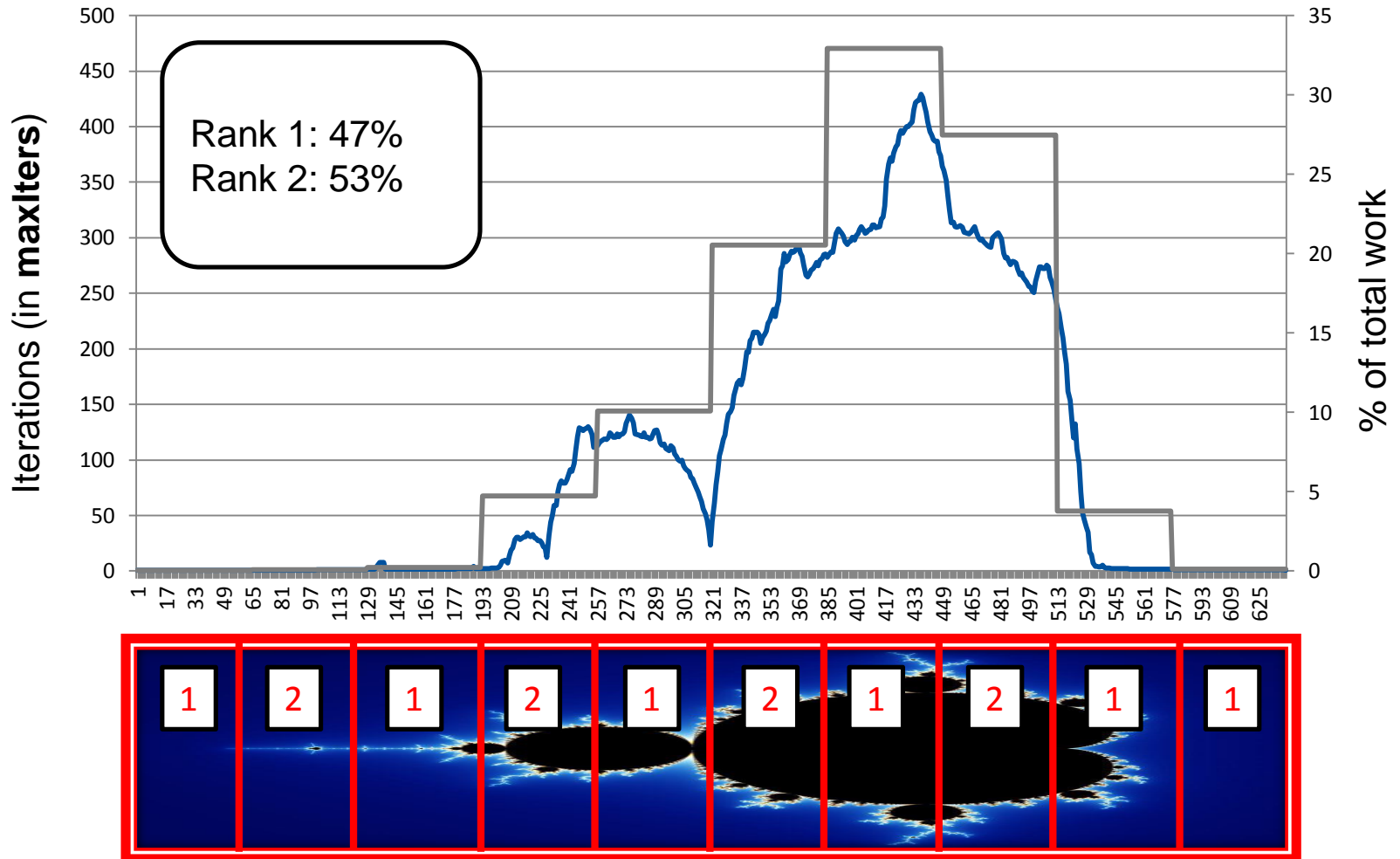
■ Controller – Worker with 3 worker processes



■ Computational complexity mapped to ranks



■ Computational complexity mapped to ranks



■ Sample implementation – controller

```
while (tiles > 0) {
    // Get a message from any worker process
    MPI_Recv(tile, BLOCKX*BLOCKY, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    switch (status.MPI_TAG) {
        case 0: // Work request
            if (br < MY/BLOCKY) {
                msg[0] = br; msg[1] = bc;
                MPI_Send(msg, 2, MPI_INT, status.MPI_SOURCE, TAG_WORK, MPI_COMM_WORLD);
                // Increment next work block pointer
                if (MX/BLOCKX == ++bc) { br++; bc = 0; }
            }
            break;
        default:
            // Worker encodes block coordinates in the tag like 1+br*(MX/BLOCKX)+bc
            resbr = (status.MPI_TAG-1) / (MX/BLOCKX);
            resbc = (status.MPI_TAG-1) % (MX/BLOCKX);
            tile_paste(matrix, resbc*BLOCKX, resbr*BLOCKY, MX, tile, BLOCKX, BLOCKY);
            tiles--;
    }
}
// All work blocks have been processed - order workers to terminate
while (size > 1) {
    MPI_Send(msg, 2, MPI_INT, size-1, TAG_TERM, MPI_COMM_WORLD);
    size--;
}
```

■ Sample implementation – worker

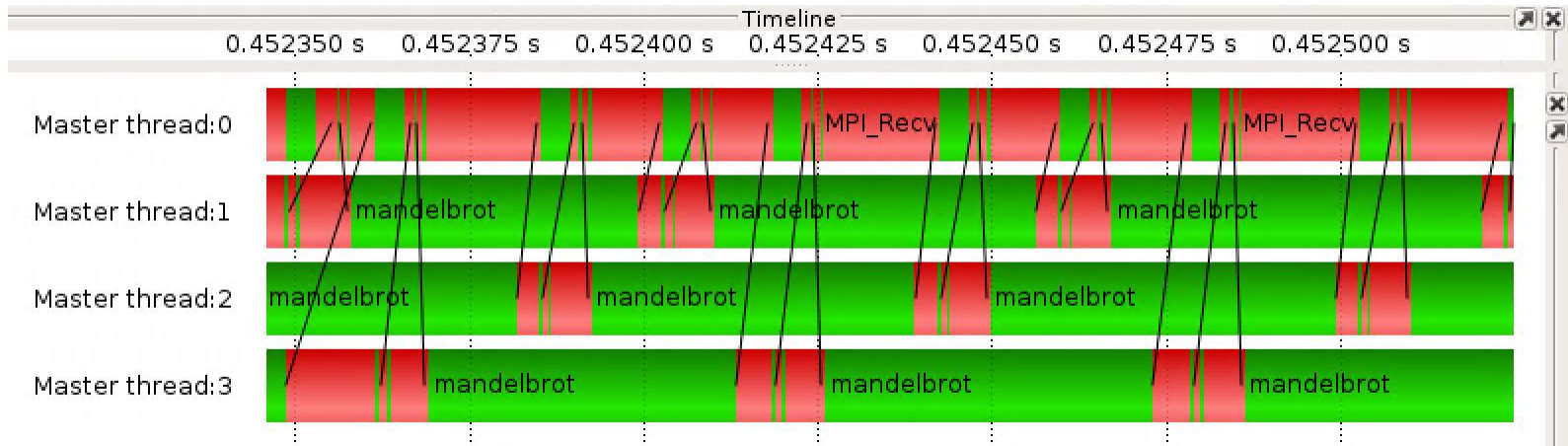
```
while (1) {
    // Request work
    MPI_Send(msg, 0, MPI_INT, 0, 0, MPI_COMM_WORLD);
    MPI_Recv(msg, 2, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    // Time to terminate?
    if (status.MPI_TAG == TAG_TERM)
        break;

    br = msg[0];
    bc = msg[1];
    iters += mandelbrot(tile, BLOCKX, BLOCKY, bc*BLOCKX, br*BLOCKY,
                       MXMIN, MXMAX, MX, MYMIN, MYMAX, MY);

    // Send result back
    MPI_Send(tile, BLOCKX*BLOCKY, MPI_INT, 0, 1+br*(MX/BLOCKX)+bc,
             MPI_COMM_WORLD);
}
```


■ Sample implementation – visualisation



- Trace information collected using Score-P and visualised with Vampir
- Sending a separate message to request for work is inefficient
 - Could be implemented better

■ **Static work distribution:**

- Works best for regular problems
- Very simple to implement (e.g. nothing really to implement)
- Irregular problems result in work imbalance (e.g. Mandelbrot)
- Not usable for dynamic workspace problems (e.g. adaptive integration)

■ **Controller – Worker approach:**

- Allows for great implementation flexibility
- Automatic work balancing if work units are properly sized
- Can be used for problems with dynamic workspaces
- Performs well on heterogeneous systems (e.g. CoWs)



■ Here is a list of important MPI topics not covered by this course:

- Parallel I/O
- Neighbour (Sparse) Collectives (MPI-3 feature)
- Non-blocking Collectives (MPI-3 feature)
- One-sided operations (RMA)

■ And a list of more exotic topics:

- Dynamic process control
- Client/server programming style

Thank you for your attention!