Erlangen Regional Computing Center

UNIVERSITÄT GREIFSWALD
Wissen lockt. Seit 1456

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Winter term 2020/2021

Parallel Programming with OpenMP and MPI

Dr. Georg Hager
Erlangen Regional Computing Center (RRZE) at Friedrich-Alexander-Universität Erlangen-Nürnberg
Institute of Physics, Universität Greifswald

Lecture 9: More MPI – point-to-point communication

High Performance Computing

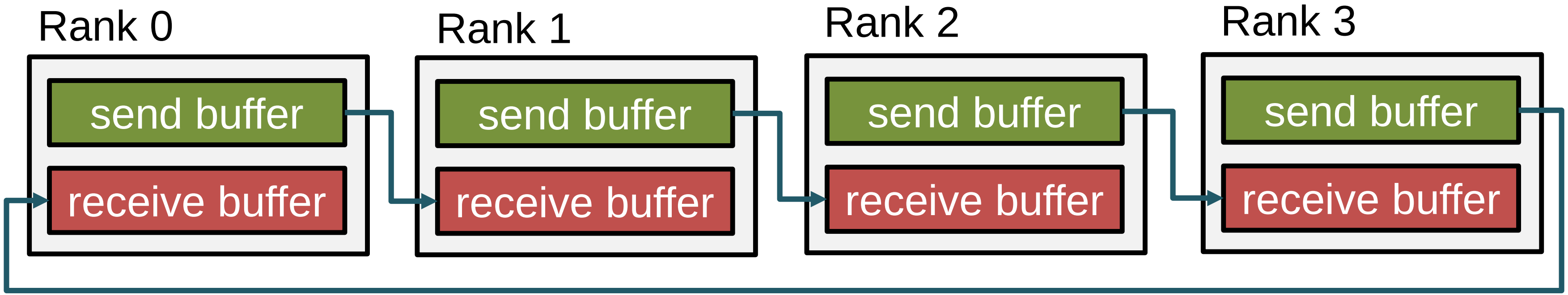
Outline of course

- Basics of parallel computer architecture
- Basics of parallel computing
- Introduction to shared-memory programming with OpenMP
- OpenMP performance issues
- Introduction to the Message Passing Interface (MPI)
- Advanced MPI
- MPI performance issues
- Hybrid MPI+OpenMP programming

Blocking point-to-point communication

Use case: Next-neighbor communication

- Frequent pattern in message passing: ring shift



- Simplistic send/recv pairing is not reliable:

```
// my left neighbor
left = (rank - 1 + size) % size;
// my right neighbor
right = (rank + 1) % size;

MPI_Send(buffer_send, n, MPI_INT, right, 1,
   MPI_COMM_WORLD);
MPI_Recv(buffer_recv, n, MPI_INT, left, 1,
         MPI_COMM_WORLD, status);
```

A simple experiment

```
// Common use case: next-neighbor data exchange
int dst;
if (rank == 0) { dst = 1; } else { dst = 0; }
char * buffer = malloc(count * sizeof(char));
MPI_Send(buffer, count, MPI_CHAR, dst, 0, MPI_COMM_WORLD);
MPI_Recv(buffer, count, MPI_CHAR, dst, 0, MPI_COMM_WORLD,
                                          MPI_STATUS_IGNORE);

$ # tested on SuperMIC@LRZ
$ mpiexec -n 2 ./send 10      # OK
$ mpiexec -n 2 ./send 100     # OK
$ mpiexec -n 2 ./send 1000    # OK
$ mpiexec -n 2 ./send 10000   # OK
$ mpiexec -n 2 ./send 100000  # OK
$ mpiexec -n 2 ./send 1000000 # DEADLOCK
```

The two variants of `MPI_Send`

Standard send is either buffered or synchronous, depending on the message size

Buffered send

- Always successful
- Time of delivery unknown
- Completion does not (necessarily) involve receiver

- Explicit call: `MPI_Bsend()`

Synchronous send

- Completion if receive operation on other end has started
- Handshake → synchronization with receiver

- Explicit call: `MPI_Ssend()`

Blocking point-to-point communication

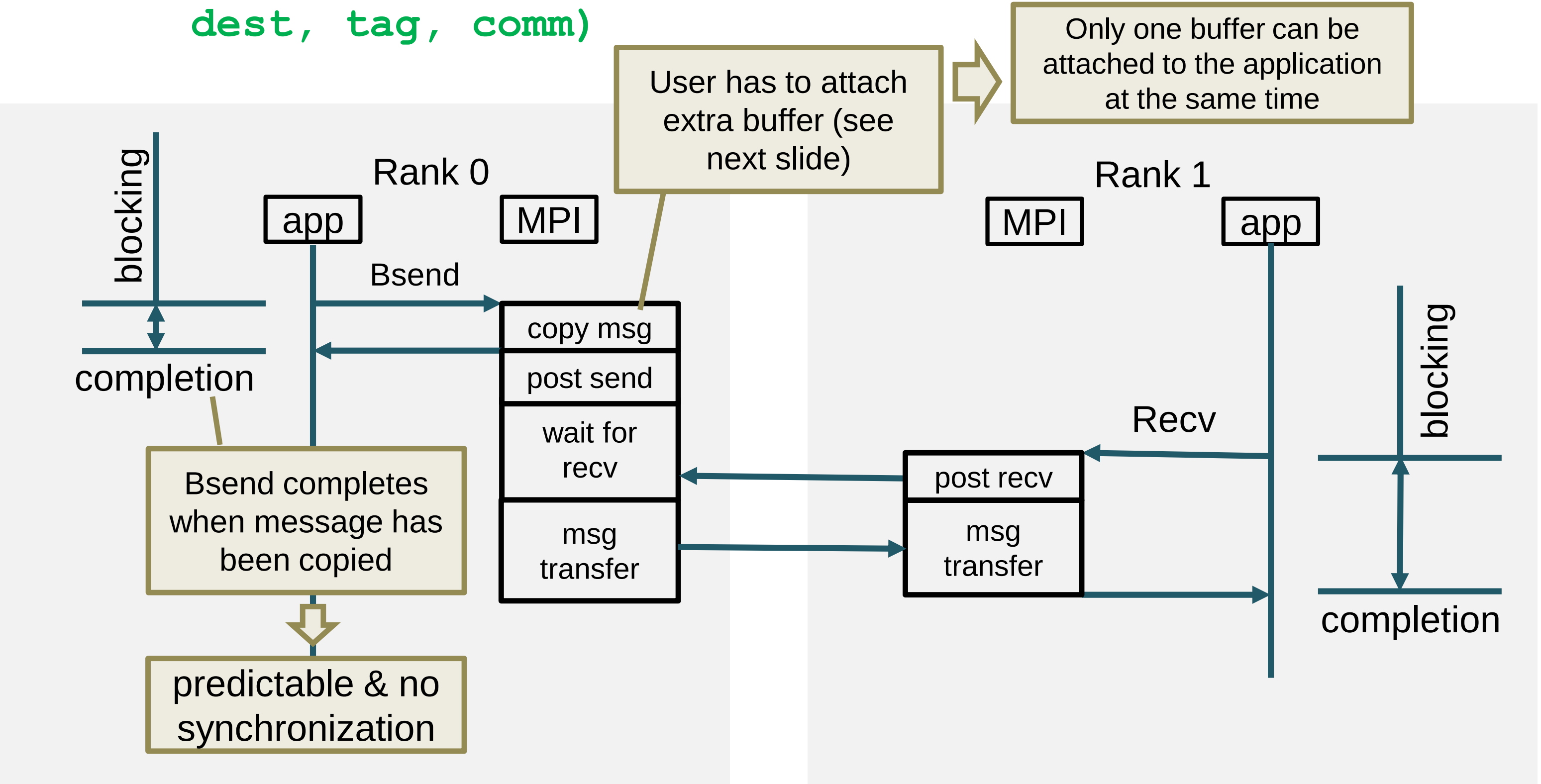
- Upon completion:
  - Buffer can be reused safely (without interfering with message transmission)
- Variants of (common) send and receive calls:

MPI function	type	completes when
`MPI_Send`	synchronous or buffered	depends on type
`MPI_Bsend`	buffered	buffer has been copied
`MPI_Ssend`	synchronous	remote starts receive
`MPI_Recv`	--	message was received

Buffered send

Caveat: comes at the cost of additional copy operations

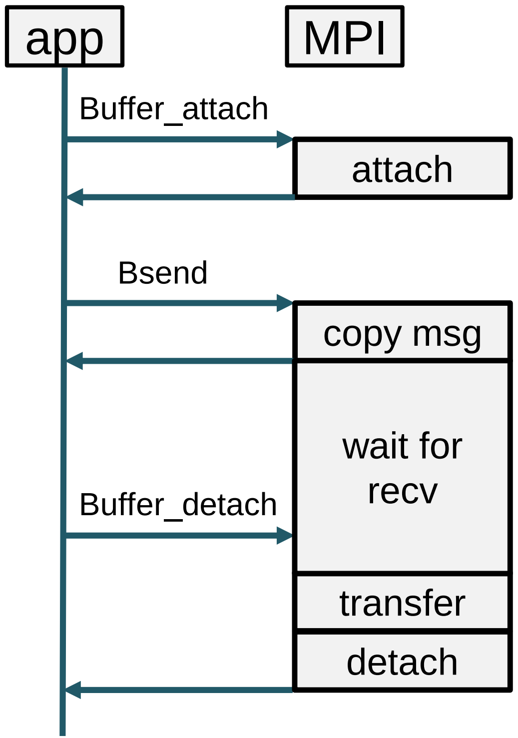
```
MPI_Bsend(buf, count, datatype,
          dest, tag, comm)
```



Attaching a buffer

- **MPI_Buffer_attach(void * buffer, int size);**
  buffer: address of buffer
  size: buffer size in bytes

  MPI_Buffer_detach(void ** buffer, int * size);
  buffer: returns addr. of detached buffer,
  defined as void *, but actually expects void **
  size: returns size of the detached buffer



- Size of buffer = (size of all outstanding BSENDs) + (number of intended BSENDs * **MPI_BSEND_OVERHEAD**)
- Best way to get required size for one message:

```
MPI_Pack_size(int incount, MPI_Datatype
datatype, MPI_Comm comm, int * s)
size = s + MPI_BSEND_OVERHEAD
```

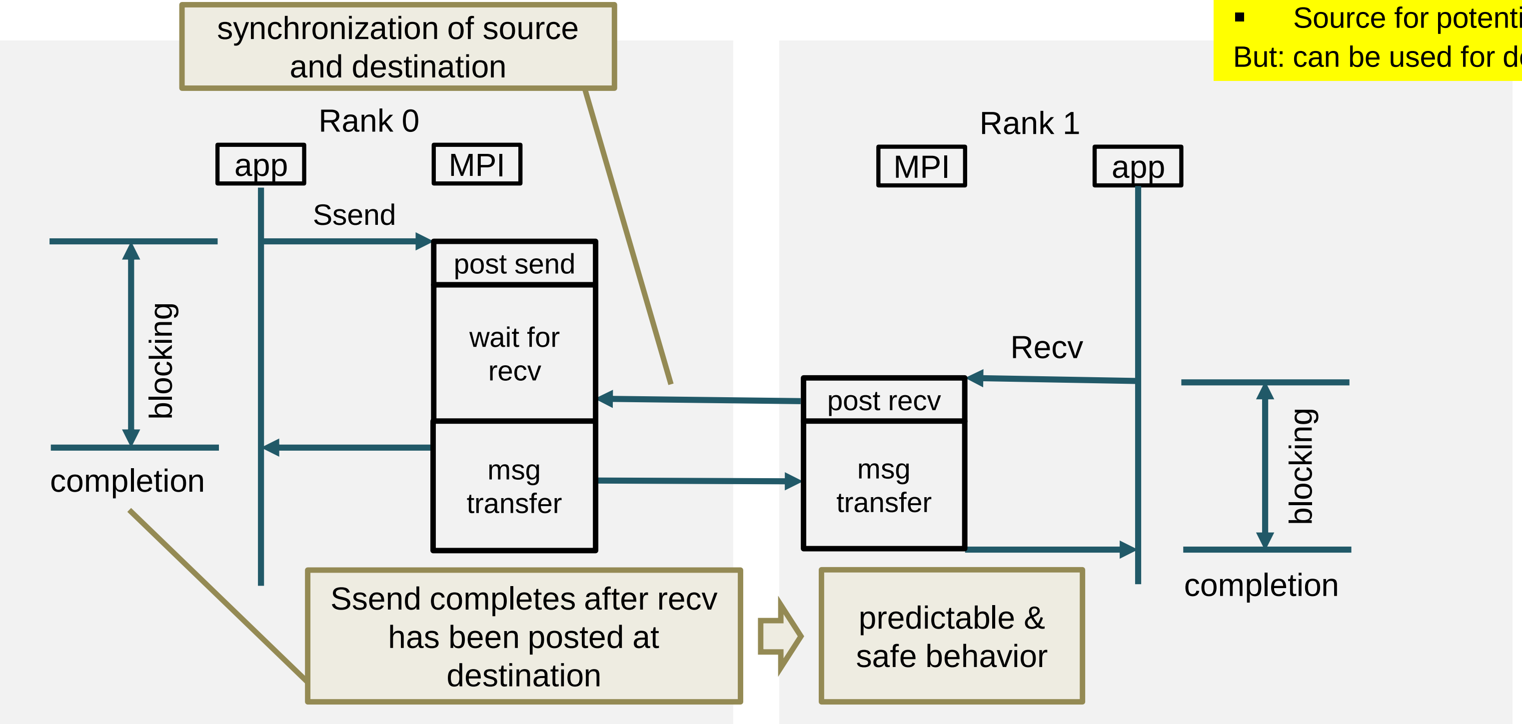
Synchronous send

```
MPI_Ssend(buf, count, datatype,
          dest, tag, comm)
```

Problems:

- Performance: high latency, risk of serialization
- Source for potential deadlocks

But: can be used for debugging

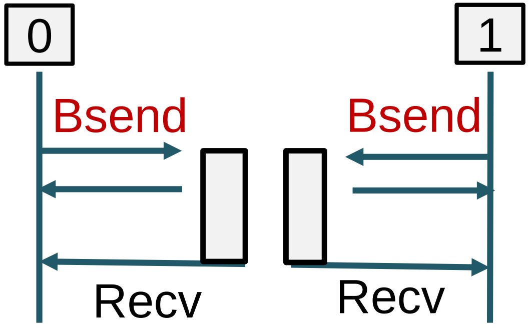


Possible solutions for the deadlock situation

MPI_Bsend: provided internal buffer takes care of everything

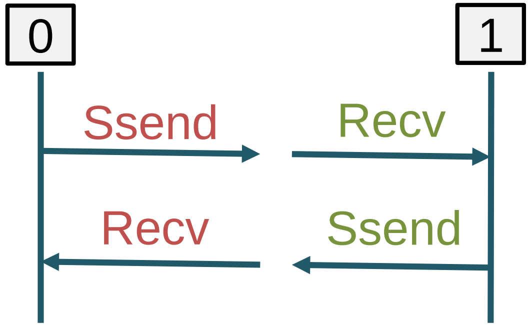
```
int dst; if (rank == 0) { dst = 1; } else { dst = 0; }
char * buffer = malloc(count * sizeof(char));

MPI_Bsend(buffer, count, MPI_CHAR, dst, 0, MPI_COMM_WORLD);
MPI_Recv(buffer, count, MPI_CHAR, dst, 0, MPI_COMM_WORLD,
                   MPI_STATUS_IGNORE);
```



MPI_Ssend: ensure matching send/receive pairs by choosing right order

```
int dst; if (rank == 0) { dst = 1; } else { dst = 0; }
char * buffer = malloc(count * sizeof(char));
if (rank == 0) {
  MPI_Ssend(buffer, count, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
  MPI_Recv(buffer, count, MPI_CHAR, 1, 0, MPI_COMM_WORLD,
                   MPI_STATUS_IGNORE);
} else {
  MPI_Recv(buffer, count, MPI_CHAR, 1, 0, MPI_COMM_WORLD,
                   MPI_STATUS_IGNORE);
  MPI_Ssend(buffer, count, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
}
```



Combining send and receive: MPI_Sendrecv

- Syntax: simple combination of send and receive arguments:

```
MPI_Sendrecv(
  buffer_send, sendcount, sendtype, dest,   sendtag,
  buffer_recv, recvcount, recvtype, source, recvtag,
  comm, status);
```

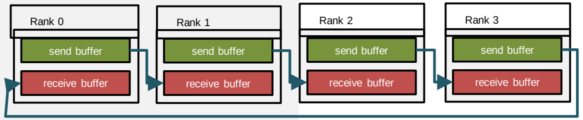
- MPI takes care that no deadlocks occur

disjoint send/receive buffers can have different count & data type

```
// my left neighbor
left = (rank - 1 + size) % size;
// my right neighbor
right = (rank + 1) % size;

MPI_Sendrecv(
        buffer_send, n, MPI_INT, right, 0,
        buffer_recv, n, MPI_INT, left,  0, MPI_COMM_WORLD, status);
```

blocking call

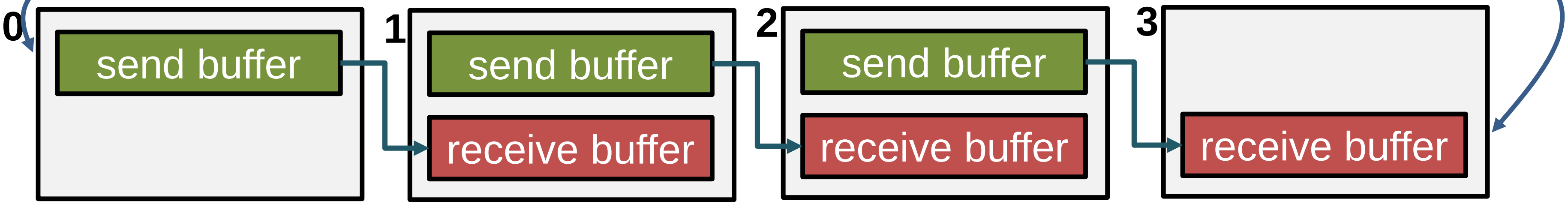


Using MPI_Sendrecv

- **MPI_Sendrecv()** matches with ***send/*recv** point-to-point calls
- **MPI_PROC_NULL** as source/destination acts as no-op
  - send/recv with **MPI_PROC_NULL** return as soon as possible, buffers are not altered
- Useful for open chains/non-circular shifts:

```
left = rank - 1; if (left < 0) { left = MPI_PROC_NULL; }
right = rank + 1; if (right >= size) {right = MPI_PROC_NULL; }

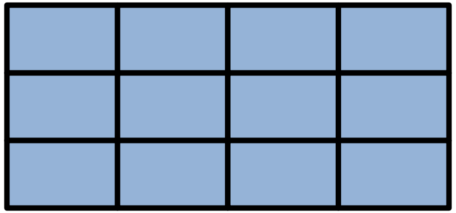
MPI_Sendrecv(
        buffer_send, n, MPI_INT, right, 0,
        buffer_recv, n, MPI_INT, left,  0, MPI_COMM_WORLD, &status);
```



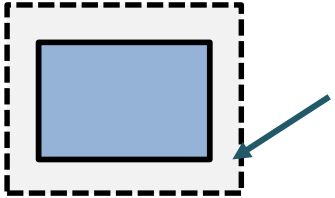
Pattern: ghost cell exchange

Many iterative algorithms require exchange of domain boundary layers

2D domain distributed to ranks (here 4 x 3), each rank gets one tile

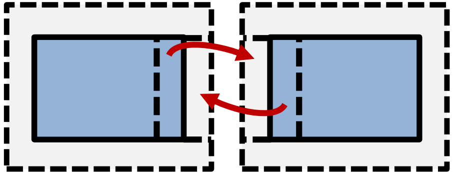


Each rank’s tile is surrounded by ghost cells, representing the cells of the neighbors



ghost cells

After each sweep over a tile, perform ghost cell exchange, i.e., update ghost cells with new values of neighbor cells

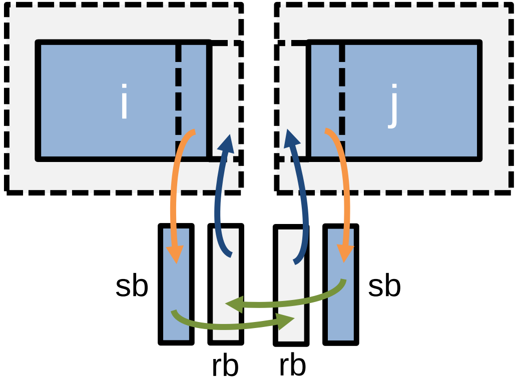


Possible implementation:

1. copy new data into contiguous send buffer
2. send to corresponding neighbor receive new data from same neighbor
3. copy new data into ghost cells

```
MPI_Sendrecv(
sb, …, j,
rb, …, j, …)
```

step 2




```
MPI_Sendrecv(
sb, …, i,
rb, …, i, …)
```

step 2

In-place communication: `MPI_Sendrecv_replace()`

- When only one single buffer is required:

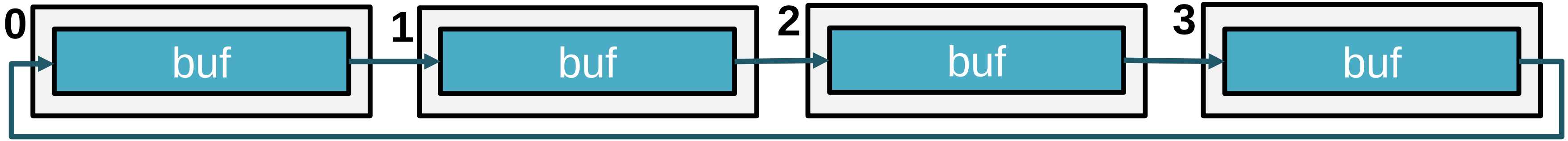
```
MPI_Sendrecv_replace(
   buf, count, datatype,
   dest, sendtag, source, recvtag,
   comm, status);
```

Same buffer, count, data type for send & receive

- MPI ensures that no deadlocks occur

```
left = (rank - 1 + size) % size;
right = (rank + 1) % size;

MPI_Sendrecv_replace(
   buf, n, MPI_INT, right, 0, left,  0, MPI_COMM_WORLD, &status);
```

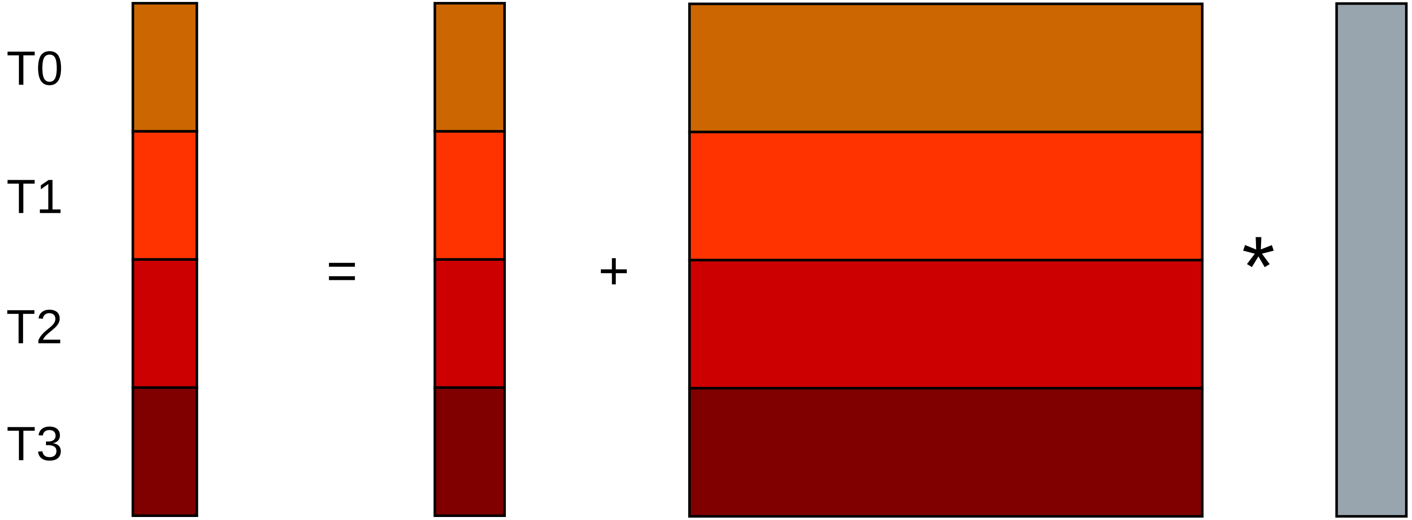


Case study: MPI-parallel dense MVM

- Remember OpenMP?

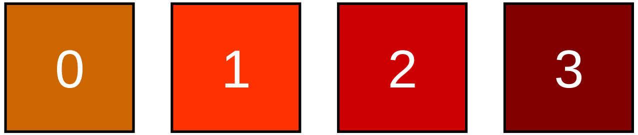
```
#pragma omp parallel for
for(int r=0; r<N; ++r)
  for(int c=0; c<N; ++c)
    y[r] += a[r][c] * x[c];
```

$$y_i = y_i + \sum_{j=1}^{N} A_{ij} x_j$$

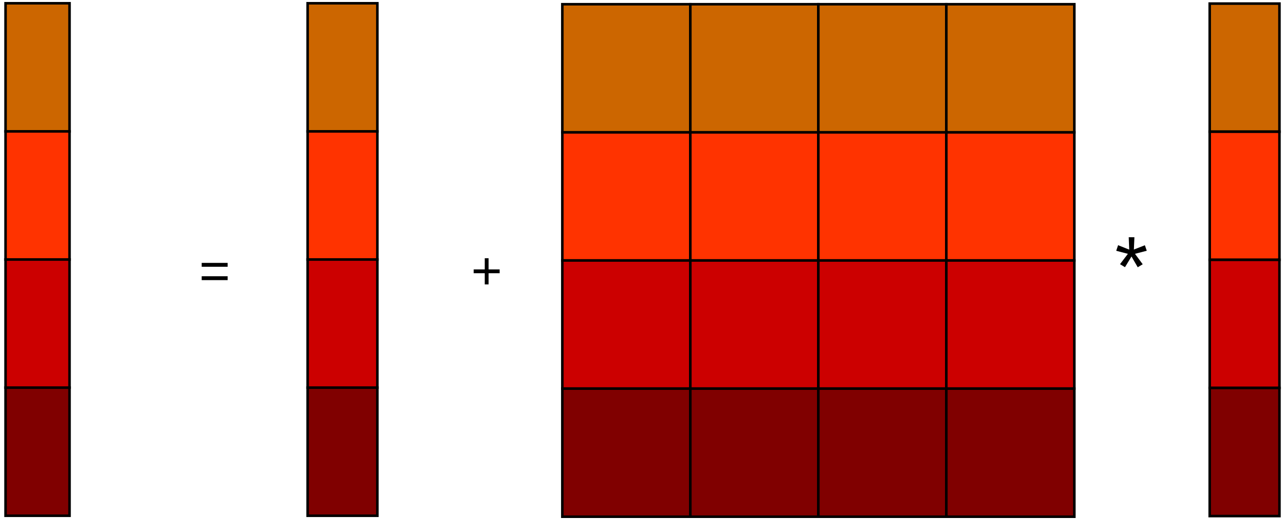


Case study: MPI-parallel dense MVM

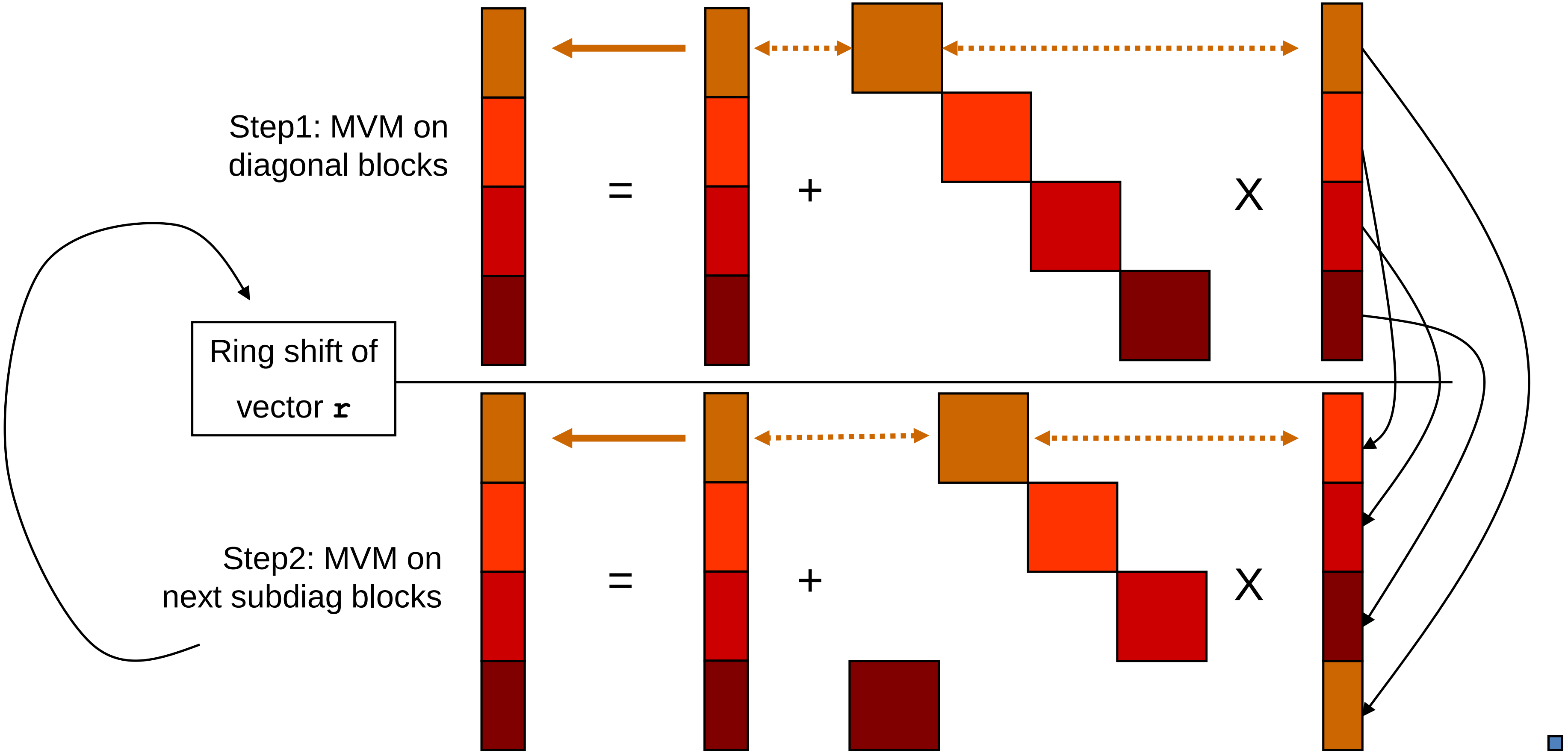
- MPI: Data distribution across ranks (matrix and vectors)



$$y_i = y_i + \sum_{j=1}^{N} A_{ij} x_j$$



MPI-parallel dense MVM



Implementation

```
int num = size / ranks; int rest = size % ranks;
l_neighbor = (rank + 1) % ranks;
r_neighbor = (rank - 1 + ranks) % ranks;
int n_start=rank*my_size+min(rest,rank), cur_size=my_size;
// loop over RHS ring shifts
for(int rot=0; rot<ranks; rot++) {
  for(int m=0; m<my_size; m++) {
    for(int n=n_start; n<n_start+cur_size; n++) {
      y[m] += a[m*size+n] * x[n-n_start];
    }
  }
  n_start += cur_size;
  if(n_start>=size) n_start=0; // wrap around
  cur_size = size_of_rank(l_neighbor,ranks,size);
  if(rot!=ranks-1) MPI_Sendrecv_replace(x, num+(rest?1:0),
                     MPI_DOUBLE, r_neighbor, 0,
                     l_neighbor, 0, MPI_COMM_WORLD, &status);
}
```

Blocking point-to-point: summary

- Blocking MPI communication calls
  - Operation locally complete when call returns
  - After completion: send/receive buffer can safely be reused
- Available send communication modes:
  - Synchronous (**`MPI_Ssend`**):
    - Handshake with receiver → performance drawbacks, deadlock dangers
  - Buffered (**`MPI_Bsend`**):
    - Completes after buffer is copied at sender
    - User-provided buffer to save messages
    - Additional copy operations
  - Standard (**`MPI_Send`**):
    - Either synchronous or buffered
    - depending on message length

Erlangen Regional Computing Center

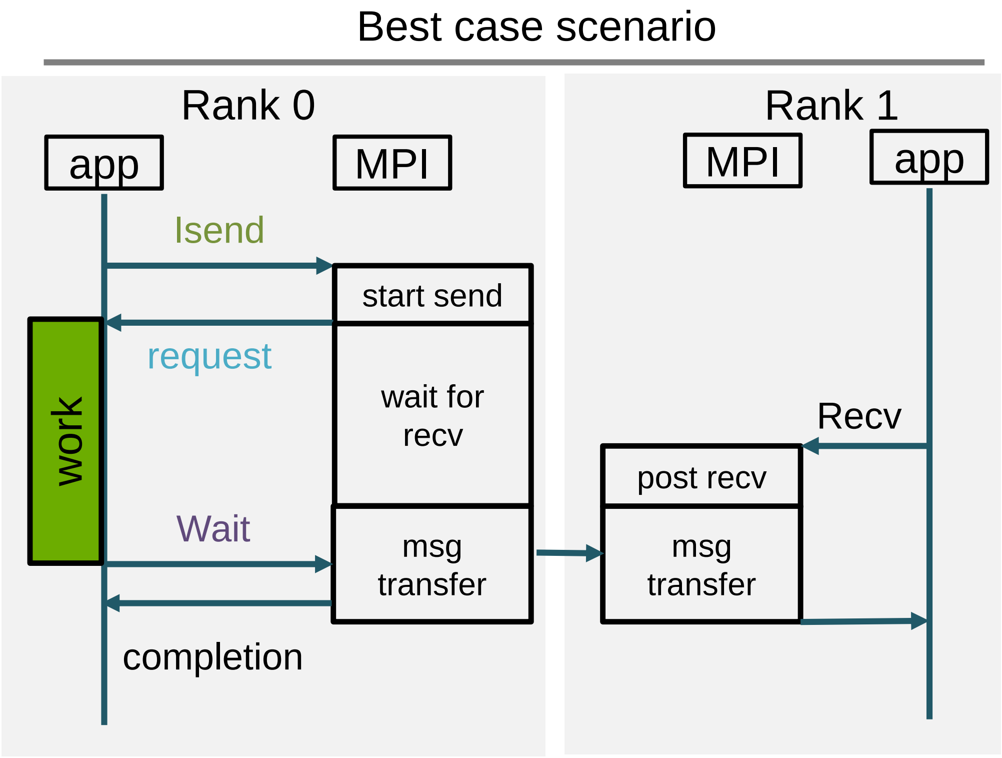
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Nonblocking point-to-point communication

High Performance Computing

Nonblocking communication

- Opportunities
  - Avoiding deadlocks
  - Opportunity for truly bidirectional communication
  - Avoid idle time
  - Avoid synchronization
  - Opportunity for overlapping communication with useful work



Standard nonblocking send/receive

- ```
  MPI_Isend(sendbuf, count, datatype, dest,   tag,
            comm, MPI_Request * request);

  MPI_Irecv(recvbuf, count, datatype, source, tag,
            comm, MPI_Request * request);
  ```

  `request`: pointer to variable of type **`MPI_Request`**,
  will be associated with the corresponding operation
- **Do not reuse `sendbuf/recvbuf` before `MPI_Isend/MPI_Irecv` has been completed**
  - Return of call does not imply completion
- **`MPI_Irecv`** has no status argument
  - obtained later during completion via **`MPI_Wait*/MPI_Test*`**
  ```

Nonblocking send and receive variants

- Completion
  - Return of `MPI_I*` call does not imply completion
  - Check for completion via `MPI_Wait* / MPI_Test*`
  - Semantics identical to blocking call after successful completion

nonblocking MPI function	blocking MPI function	type	completes when
`MPI_Isend`	`MPI_Send`	synchronous or buffered	depends on type
`MPI_Ibsend`	`MPI_Bsend`	buffered	buffer has been copied
`MPI_Issend`	`MPI_Ssend`	synchronous	remote starts receive
`MPI_Irecv`	`MPI_Recv`	--	message was received

Test for completion

Two test modes:

- Blocking
  - **`MPI_Wait*`**: Wait until the communication has been completed and buffer can safely be reused
- Nonblocking
  - **`MPI_Test*`**: Return true (false) if the communication has (not) completed

Despite the naming, the modes both pertain to nonblocking point-to-point communication!

Test for completion – single request

- Test one communication handle for completion:

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

request: request handle of type **MPI_Request**

status: status object of type **MPI_Status** (cf. **MPI_Recv**)

flag: variable of type **int** to test for success

Use of wait/test

MPI_Wait

```
MPI_Request request;
MPI_Status status;


MPI_Isend(
  send_buffer, count, MPI_CHAR,
  dst, 0, MPI_COMM_WORLD, &request);


// do some work…
// do not use send_buffer

MPI_Wait(&request, &status);

// use send_buffer
```

MPI_Test

```
MPI_Request request;
MPI_Status status;
int flag;

MPI_Isend(
  send_buffer, count, MPI_CHAR,
  dst, 0, MPI_COMM_WORLD, &request);

do {
    // do some work…
    // do not use send_buffer
    MPI_Test(&request, &flag, &status);
} while (!flag);

// use send_buffer
```

Wait for completion – all requests in a list

- MPI can handle multiple communication requests
- Wait/Test for completion of multiple requests:

```
MPI_Waitall(int count, MPI_Request requests[],
                       MPI_Status statuses[]);

MPI_Testall(int count, MPI_Request requests[],
            int *flag, MPI_Status statuses[]);
```

- Waits for/Tests if all provided requests have been completed

Use of `MPI_Waitall`

```
MPI_Request requests[2];
MPI_Status  statuses[2];

MPI_Isend(send_buffer, …, &(requests[0]));
MPI_Irecv(recv_buffer, …, &(requests[1]));

// do some work…

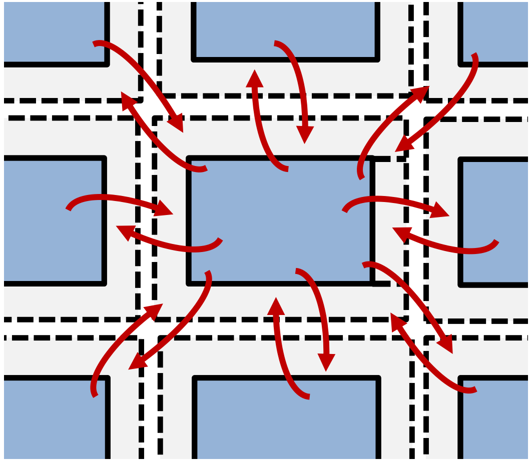
MPI_Waitall(2, requests, statuses)
// Isend & Irecv have been completed
```

Arrays of requests and statuses

number of elements in the arrays

Ghost cell exchange with nonblocking MPI

Ghost cell exchange with nonblocking send/recv with all neighbors at once



Possible implementation:

1. Copy new data into contiguous send buffers
2. Start nonblocking receives/sends from/to corresponding neighbors
3. Update local cells that do not need halo cells for boundary conditions (“bulk update”)
4. Wait with MPI_Waitall for all obtained requests to complete
5. Copy received halo data into ghost cells
6. Update cells that need the halo

→ Opportunity to overlap communication with bulk update (MPI implementation permitting)

Wait for completion – one or several requests out of a list

Wait for/Test if exactly one request among many has been completed

```
MPI_Waitany(int count, MPI_Request requests[],
            int * idx, MPI_Status * status);

MPI_Testany(int count, MPI_Request requests[],
            int * idx, int * flag,
            MPI_Status * status);
```

Wait for/Test if at least one request among many has been completed

```
MPI_Waitsome(int incount, MPI_Request requests[], int * outcount,
             int indices[], MPI_Status statuses[]);

MPI_Testsome(int incount, MPI_Request requests[], int * outcount,
             int indices[], MPI_Status statuses[]);
```

Use of MPI_Testany

```
MPI_Request requests[2];
MPI_Status  status;
int finished = 0;

MPI_Isend(send_buffer, …, &(requests[0]));
MPI_Irecv(recv_buffer, …, &(requests[1]));
do {
  // do some work…
  MPI_Testany(2, requests, &idx, &flag, &status);
  if (flag) { ++finished; }
} while (finished < 2);
```

- completed requests are automatically set to `MPI_REQUEST_NULL`
- completed requests: `requests[idx]`

Pitfalls with nonblocking MPI and compiler optimizations

- Fortran:
```
MPI_IRECV(recvbuf, ..., request, ierror)
MPI_WAIT(request, status, ierror)
write (*,*) recvbuf
```
- may be compiled as
```
MPI_IRECV(recvbuf, ..., request, ierror)
registerA = recvbuf
MPI_WAIT(request, status, ierror)
write (*,*) registerA
```

MPI might modify recvbuf after MPI_IRECV returns, but the compiler has no idea about this

- i.e., old data is written instead of received data!

- Workarounds:
  - **recvbuf** may be allocated in a common block, or
  - calling **MPI_GET_ADDRESS(recvbuf, iaddr_dummy, ierror)** after **MPI_WAIT**

Nonblocking point-to-point communication

- Standard nonblocking send/recv `MPI_Isend()`/`MPI_Irecv()`
  - Return of call does not imply completion of operation
  - Use `MPI_Wait*()` / `MPI_Test*()` to check for completion using request handles
- All outstanding requests must be completed!
- Potentials
  - Overlapping of communication with work (not guaranteed by MPI standard)
  - Overlapping send and receive
  - Avoiding synchronization and idle times
- Caveat: Compiler does not know about asynchronous modification of data