# MPI: Part II

## AQTIVATE Training Workshop I


Dirk Pleiter

CST | EECS | KTH



December 2023

# Overview

Buffering and Non-Blocking Communication

Halo Exchange

Collective Communication

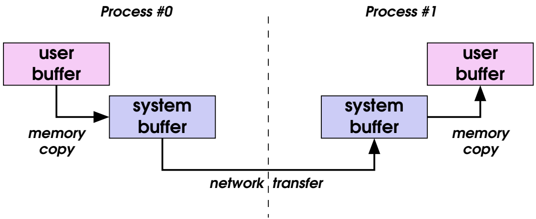
# Content

# Communication Buffering (1/3)

- MPI communication involves multiple transactions
- Challenges:
  - Intermediate buffer space required
  - Buffered data must remain unchanged until buffer is completely read

# Communication Buffering (2/3)

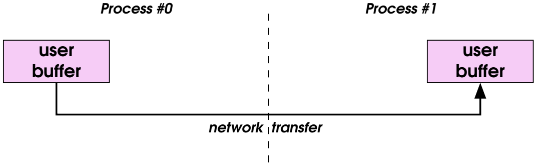
- Design option #1: Intermediate memory buffer




  - Advantages:
    - MPI send completes after fast in-memory copy
  - Disadvantages:
    - Need additional space in memory
    - Performing memory copies increases pressure on memory bus

# Communication Buffering (3/3)

- Design option #2: Zero-copy communication




- Advantages:
  - No additional memory space and less memory traffic
- Disadvantages:
  - MPI send only completes once data has been transferred over network

# MPI Buffered Mode

- User can explicitly request communication to be buffered (**buffered mode**):

```c
int MPI_Bsend(
  const void* buf,           /* Pointer to send buffer */
  int count,                 /* Number of elements */
  MPI_Datatype datatype,     /* Data type */
  int dest,                  /* Destination rank */
  int tag,                   /* Communication tag */
  MPI_Comm comm              /* Communicator */
);
```

- A buffered mode send operation can be started whether or not a matching receive has been posted

# MPI Buffered Mode: User-provided Buffer

- A user may specify a buffer to be used for buffering messages sent in buffered mode:

```
int MPI_Buffer_attach(
 void* sysbuf,    /* Pointer to user allocated system buffer */
 int size         /* Size of buffer in Bytes */
);

int MPI_Buffer_detach(
  void* sysbuf,   /* Returned pointer to system buffer */
  int* size       /* Returned size of buffer */
);
```

# Blocking versus Non-Blocking Communication

- Blocking
  - Function returns only after completion of the associated operation
  - Examples: `MPI_Send`, `MPI_Recv`
- Non-blocking
  - Function may return before the associated operation has completed
  - Examples:
    - Immediate send: `MPI_Isend`
    - Immediate receive: `MPI_Irecv`
  - Resources (e.g. message buffers) passed to function must not be reused until the operation has completed
    - Non-blocking functions return a handle that allow to query status of the initiated operation

# MPI Immediate Send and Receive

- Syntax of immediate send operation

```c
int MPI_Isend(
 const void* buf,              /* Send buffer */
 int count,                    /* Number of elements */
 MPI_Datatype datatype,        /* Data type */
 int dest,                     /* Destination rank */
 int tag,                      /* Communication tag */
 MPI_Comm comm,                /* Communicator */
 MPI_Request *request          /* Request handle */
);
```

- Syntax of immediate receive operation

```c
int MPI_Irecv(
  const void* buf,             /* Pointer to receive buffer */
  int count,                   /* Number of elements */
  MPI_Datatype datatype,       /* Data type */
  int source,                  /* Rank of source */
  int tag,                     /* Message tag */
  MPI_Comm comm,               /* Communicator */
  MPI_Status *status,          /* Status object */
  MPI_Request *request         /* Request handle */
);
```

# Waiting for Completion (1/2)

- To wait for a communication identified by a given request handler to complete, the blocking function `MPI_Wait` can be used:

```
int MPI_Wait(
  MPI_Request *request,   /* Request handle (in/out) */
  MPI_Status *status      /* Status object (out) */
);
```

  - On return the status object is updated

- `MPI_Test` is available to perform checks without block:

```
int MPI_Test(
  MPI_Request *request,   /* Request handle (in/out) */
  int *flag,              /* Completion flag (out) */
  MPI_Status *status      /* Status object (out) */
);
```

  - The returned `flag` is set to true when operation completed

# Waiting for Completion (2/2)

- Use `MPI_Waitall` when waiting for a set of requests to complete:

```c
int MPI_Waitall(
  int count,                /* Array size */
  MPI_Request request[],    /* Array of requests */
  MPI_Status status[]       /* Array of status objects */
);
```

# MPI Modes Overview (1/2)

- **Standard mode:**
  - A send operation is started whether or not a matching receive has been posted
  - It may complete before a matching receive is posted
- **Synchronous mode:**
  - A send operation can be started whether or not a matching receive has been posted
  - The send operation send will complete successfully only if a matching receive is posted
- **Ready mode:**
  - A send operation may be started only if the matching receive is already posted
- **Buffered mode:**
  - A send operation can be started whether or not a matching receive has been posted
  - It may complete before a matching receive is posted

# MPI Modes Overview (2/2)

The following variants of MPI send are available:

| Mode | Blocking variant | Non-blocking variant |
|---|---|---|
| Standard | `MPI_Send` | `MPI_Isend` |
| Synchronous | `MPI_SSend` | `MPI_Issend` |
| Ready | `MPI_RSend` | `MPI_Irsend` |
| Buffered | `MPI_BSend` | `MPI_Ibsend` |

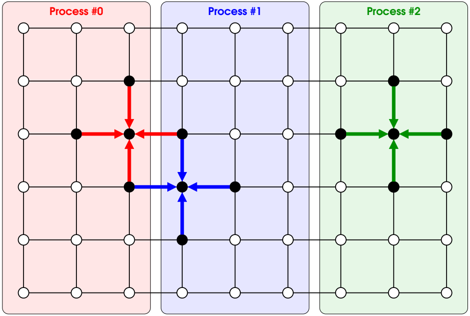
# Non-Blocking Collectives

- Collective operations are typically also available in a non-blocking variant
  - Examples
    - `MPI_Ibcast`
    - `MPI_Ireduce`
- These functions return a request handle such that `MPI_Wait` or `MPI_test` can be used to check for completion

# Content

# Solving the 2D Poisson in Parallel

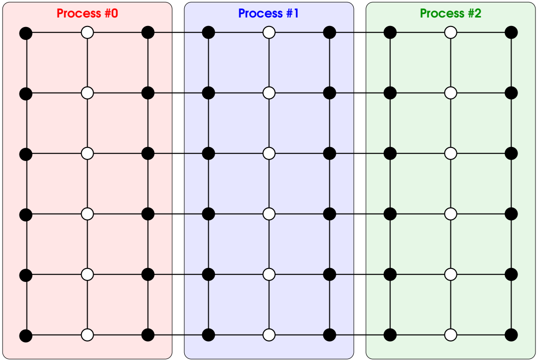
- Graphical representation of a parallel version of the discrete 2-dimensional Poisson equation:




- Local update may require data from remote process

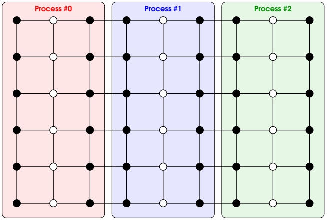
# 2D Poisson: Halo versus Bulk

- Classification of grid points:
  - Inner points (bulk): No data from other processes needed
  - Boundary points (halo): Data from other processes needed
- Parallel update requires **halo exchange**

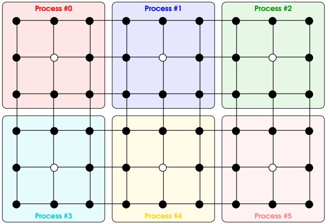


# 2D Poisson Equation: Data Decomposition

- Data decomposition in 1 dimension:




- Data decomposition in 2 dimensions:



# 2D Poisson Equation: Information Exchange

- Assume a square lattice of size $L^2$ being distributed over $P$ processes and use of double-precision numbers
- Update of each site requires 8 Flop:

$$I_{\mathrm{fp}}(L, P) = (8 \cdot L^2 / P)\,\mathrm{Flop}$$

- In each iteration, the field $v$ is updated and the local halo needs to be communicated
  - Data decomposition in 1 dimension:

$$I_{\mathrm{net}}^{(1\mathrm{d})}(L, P) = (2 \cdot L \cdot 8)\,\mathrm{Byte}$$

  - Data decomposition in 2 dimensions:

$$I_{\mathrm{net}}^{(2\mathrm{d})}(L, P) \simeq \left(4 \cdot (L/\sqrt{P} - 1) \cdot 8\right)\,\mathrm{Byte}$$

# 2D Poisson Equation: Data Decomposition Analysis

- For fixed $P$ we find

$$\frac{I_{\mathrm{fp}}}{I_{\mathrm{net}}} \propto L$$

  - Increasing $L$ the amount of computation relative to network communication increases

- For fixed $L$ we find

$$\frac{I_{\mathrm{net}}^{(1\mathrm{d})}}{I_{\mathrm{fp}}} \propto P\,, \qquad \frac{I_{\mathrm{net}}^{(2\mathrm{d})}}{I_{\mathrm{fp}}} \propto \sqrt{P}$$

  - 2-dimensional data decomposition requires less data to be communicated (for sufficiently large $L$)
  - But: In higher dimensions halo becomes fragmented in memory

# 2D Poisson Equation: Standard Send

- Consider halo update using the following pseudo-code:

```
foreach neighbour i:
  MPI_Send(..., nb[i], ...)

foreach neighbour i:
  MPI_Recv(..., nb[i], ...)
```

- Will this code work? Why not?

# 2D Poisson Equation: Immediate Send

- Answer: No, the send operations will start to block
- Fixed halo update using immediate send and receive operations:

```
foreach neighbour i:
  MPI_Irecv(..., nb[i], ..., recv_req[i])

foreach neighbour i:
  MPI_Isend(..., nb[i], ..., send_req[i])
MPI_Waitall(...)
```

# 2D Poisson Equation: Optimisations (1/2)

- Optimisation strategy #1: Overlap communication and computation
  - Steps:
    1. Initiate communication of boundary points
    2. Update interior points
    3. Wait for communications to complete
    4. Update boundary points
  - If $\Delta t(L) = \max\left[\Delta t_{\mathrm{fp}}(L), \Delta t_{\mathrm{net}}(L)\right]$ and latency-bandwidth model to hold then problem becomes local performance bound for large $L$ as

$$\Delta t \to \Delta t_{\mathrm{fp}}(L) \quad \text{for} \quad L \to \infty$$

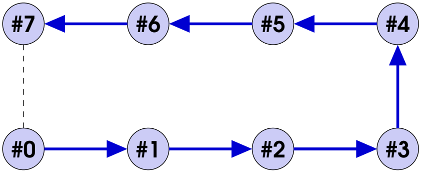
# 2D Poisson Equation: Optimisations (2/2)

- Optimisation strategy #2: Minimise number of send-receive operations
  - Solution: Copy halo into single buffer before communication
  - Alternative solution: Use advance data types (see next section)
- Performance benefits difficult to predict as message rates improved
  - But: Higher network bandwidth demands due to additional overhead (message headers, minimal message length)

# Content

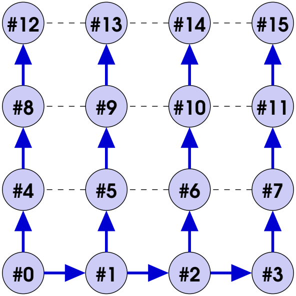
# Communication Patterns: One-to-All Broadcast

- One processor has a piece of data which needs to be sent to all other processes
- Implementation assuming ring topology with $P$ processes and process #0 being data source
- Need to perform sequence of send-receive operations until data arrives at node #P−1
  $\Rightarrow$ **Complexity** $= P - 1$ steps



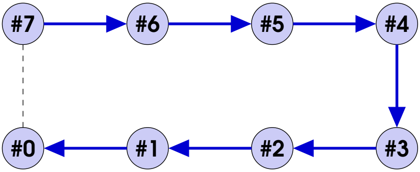
# Communication Patterns: Broadcast (2)

- Implementation on 2-dimensional mesh
  $\Rightarrow$ Complexity $\sim 2\sqrt{P}$
- Generalization to *d*-dimension mesh
  $\Rightarrow$ Complexity $\sim d\, P^{1/d}$



# Communication Patterns: All-to-One Reduce

- Data on all processes is reduced to single piece of data on one processor
- Examples for reduction operators
  - Summation
  - Selection of minimum/maximum value
- Implementation assuming ring topology with $P$ processes and process #0 being final destination:



# Communication Patterns: All-to-One Reduce (2)

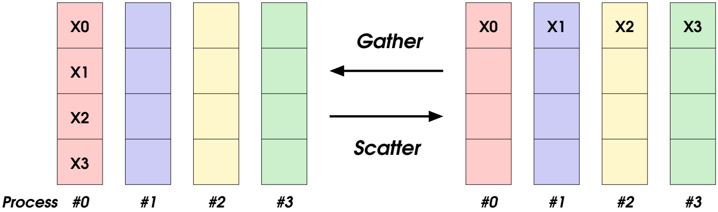
- Implementation of global sum $S$:

$$s_k = \sum_{i=k\,(N/P)}^{(k+1)(N/P)-1} x_i, \qquad S = \sum_{k=0}^{P-1} s_k$$

  with $s_k$ distributed over $P$ processes connected in a ring topology:
  - Process #P-1:
    - Send $s_{P-1}$ to process #P-2
  - Process #P-2:
    - Receive $s_{P-1}$ from process #P-1
    - Compute $s_{P-2} + s_{P-1}$
    - Send result to process #P-3
  - ...
  - Process #0:
    - Receive $\sum_{k=1}^{P-1} s_k$ from process #1
    - Compute $\sum_{k=0}^{P-1} s_k$

# Communication Patterns: Gather and Scatter

- **Scatter**: One process distributes different pieces of data to all other processes
  - Also called: one-to-all personalized communication
  - Operation fundamentally different from broadcast
- **Gather**: One process collects one piece of data from all other processes
  - Inverse of scatter operation



# MPI Collective Operations: Overview

- MPI defines 9 types of collective operations:
  - Barrier
  - Broadcast
  - Gather
  - All-Gather
  - Scatter
  - All-to-all
  - Reduce
  - All-Reduce
  - Reduce-Scatter
  - Scan
- Collective communication is over all of the processes in particular group identified by a communicator
- Standard meanwhile introduced blocking and non-blocking variants

# MPI Barrier

- Syntax:

```
int MPI_Barrier(MPI_Comm comm)

MPI_Barrier(comm, ierror)
  TYPE(MPI_Comm), INTENT(IN)      :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

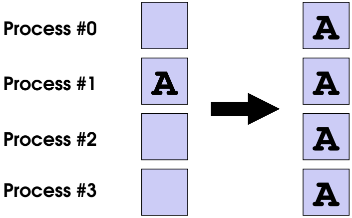
- Function blocks the caller until all group members (identified by the communicator `comm`) have called it
  - A deadlock occurs if one group member does not call the function
- Practical advice
  - In a program with send-receive and other collective communications, an MPI barrier is rarely needed
  - MPI makes no guarantees on how long it will take other processes to leave the barrier
    - MPI barriers are not appropriate for highly accurate time measurements

# MPI Broadcast

- Syntax:

```c
int MPI_Bcast(
 void* buffer,              /* Pointer to output/input buffer */
 int count,                 /* Number of elements */
 MPI_Datatype datatype,     /* Data type */
 int root,                  /* Rank of root process */
 MPI_Comm comm              /* Communicator */
);
```

- Function copies data from process identified by `root` and `comm` to all other processes of the group
- Example with 4 processes and `root=1`:

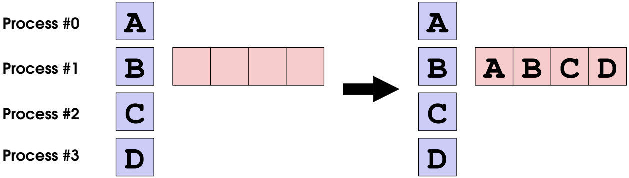


# MPI Gather

- Syntax:

```c
int MPI_Gather(
 const void* sendbuf,    /* Pointer to output buffer */
 int sendcount,          /* Number of elements */
 MPI_Datatype sendtype,  /* Data type */
 void* recvbuf,          /* Pointer to input buffer */
 int recvcount,          /* Number of elements */
 MPI_Datatype recvtype,  /* Data type */
 int root,               /* Rank of receiving process */
 MPI_Comm comm           /* Communicator */
);
```

- Function collects data on process identified by `root` and `comm` sent by all processes of the group (in rank order)

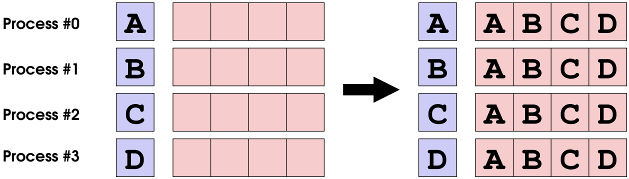


# MPI All Gather

- Syntax:

```c
int MPI_Allgather(
  const void* sendbuf,      /* Pointer to send buffer */
  int sendcount,            /* Number of elements */
  MPI_Datatype sendtype,    /* Data type */
  void* recvbuf,            /* Pointer to recv buffer */
  int recvcount,            /* Number of elements */
  MPI_Datatype recvtype,    /* Data type */
  MPI_Comm comm             /* Communicator */
);
```

- Function similar to `MPI_Gather`, but now all processes receive the data

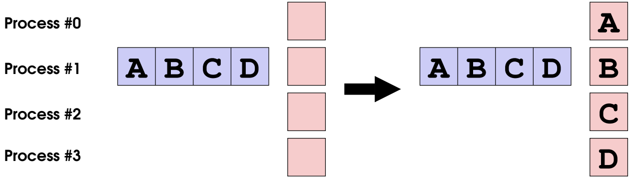


# MPI Scatter

- Syntax:

```
int MPI_Scatter(
  const void* sendbuf,      /* Pointer to output buffer */
  int sendcount,            /* Number of elements */
  MPI_Datatype sendtype,    /* Data type */
  void* recvbuf,            /* Pointer to input buffer */
  int recvcount,            /* Number of elements */
  MPI_Datatype recvtype,    /* Data type */
  int root,                 /* Rank of sending root process */
  MPI_Comm comm             /* Communicator */
);
```

- This function is the inverse operation to `MPI_Gather`

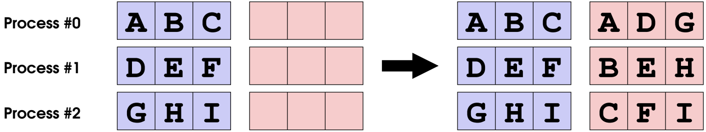


# MPI All-to-all

- Syntax:

```c
int MPI_Alltoall(
  const void* sendbuf,        /* Pointer to output buffer */
  int sendcount,              /* Number of elements */
  MPI_Datatype sendtype,      /* Data type */
  void* recvbuf,              /* Pointer to input buffer */
  int recvcount,              /* Number of elements */
  MPI_Datatype recvtype,      /* Data type */
  MPI_Comm comm               /* Communicator */
);
```

- Each process sends distinct data to each of the other processes within the group

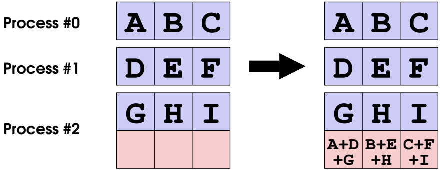


# MPI Reduce (1/2)

- Syntax:

```c
int MPI_Reduce(
  const void* sendbuf,      /* Pointer to output buffer */
  void* recvbuf,            /* Pointer to input buffer */
  int count,                /* Number of elements */
  MPI_Datatype datatype,    /* Data type */
  MPI_Op op,                /* Reduction operator */
  int root,                 /* Rank of root process */
  MPI_Comm comm             /* Communicator */
);
```

- Function combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank `root`



# MPI Reduce (2/2)

- MPI's predefined reduction operations:

| Name | Meaning |
|---|---|
| MPI_MAX | maximum |
| MPI_MIN | minimum |
| MPI_SUM | sum |
| MPI_PROD | product |
| MPI_LAND | logical and |
| MPI_BAND | bit-wise and |
| MPI_LOR | logical or |
| MPI_BOR | bit-wise or |
| MPI_LXOR | logical exclusive or (xor) |
| MPI_BXOR | bit-wise exclusive or (xor) |
| MPI_MAXLOC | max value and location |
| MPI_MINLOC | min value and location |

# MPI Reduce Example: Global Sum

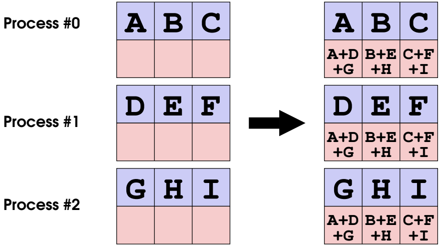
```c
int main() {
    const int N=1048576;
    int isize, irank;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &irank);
    MPI_Comm_size(MPI_COMM_WORLD, &isize);

    int n = N / isize;   /* Assume N to be multiple of isize */
    int *v = (int *) calloc(n, sizeof(int));

    for (int i = 0; i < n; i++) v[i] = irank * n + i + 1;

    double s = 0;        /* Local sum */
    for (int i = 0; i < n; i++) s += v[i];

    double gs = -1;      /* Global sum */
    MPI_Reduce(&s, &gs, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (irank == 0) printf("N=%d, gs=%.2f\n", N, gs);

    free(v);
    MPI_Finalize();

    return 0;
}
```

# MPI All-Reduce

- Syntax:

```
int MPI_Allreduce(
  const void* sendbuf,       /* Pointer to output buffer */
  void* recvbuf,             /* Pointer to input buffer */
  int count,                 /* Number of elements */
  MPI_Datatype datatype,     /* Data type */
  MPI_Op op,                 /* Reduction operator */
  MPI_Comm comm              /* Communicator */
);
```

- Similar to `MPI_Reduce` but now all processes will receive an identical copy of the result



# MPI Collectives: Caveats

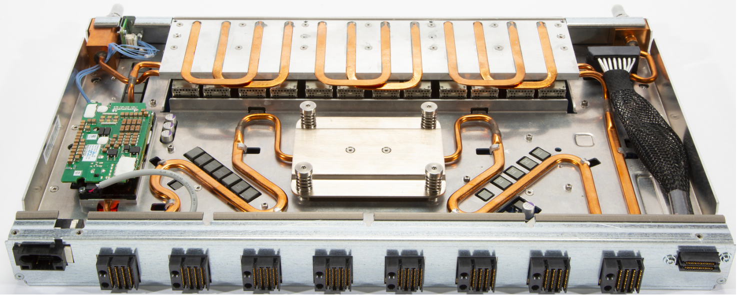
- Collective operations have to be called by all processes within the given group
- Collective operations have to be issued in the same order by all participating processes
- Computations of collectives may not be deterministic
  - In exact arithmetic always the same result will be produce
  - In case of floating-point numbers rounding errors may lead to non-deterministic results
    - Re-call that in floating-point arithmetic operations may not be commutative, i.e. $(a + b) + c \neq (c + b) + a$
  - The MPI standard does not require that the same input give the same output every time
    - Exact results are not relevant for all applications
    - No having such a requirement opens performance opportunities, e.g. by moving computations in the network such that order of the arithmetic operations will depend on network topology

# MPI Collectives: Optimisations

- For many operations, the predefined constant `MPI_IN_PLACE` can be used as send buffer argument
  - No need to allocate and manage separate send and receive buffers

# Finish with Current Networking Technology

HPE Cray Rosetta switch with 64 ports:



[Wikichip]