



Non-blocking communication & performance considerations

Slides Sebastian von Alfthan and Pekka Manninen

Point to point communications

- We discussed send and receive operations enabling any parallel application
 1. Blocking: MPI_Send MPI_Recv
 2. Blocking: MPI_Sendrecv

Nonblocking communication

- Non-blocking sends and receives
 - **MPI_Isend & MPI_Irecv**
 - Returns immediately and sends/receives in background
- Enables some computing concurrently with communication
- You can mix non-blocking and blocking routines!
 - E.g., receive MPI_Isend with MPI_Recv

Nonblocking communication

- Have to finalize send/receive operations
 - **MPI_Wait, MPI_Waitall,...**
 - Waits for the communication started with MPI_Isend or MPI_Irecv to finish (blocking)
 - **MPI_Test,...**
 - Tests if the communication has finished (non-blocking)
- Typical usage pattern
 1. Start: MPI_Isend, MPI_Irecv
 2. Compute...
 3. Finish: MPI_Waitall

Non-blocking send

```
int MPI_Isend(buf, count, datatype, dest, tag,  
             comm, request )
```

- Parameters
 - Similar to MPI_Send
 - **buf** Send buffer shall not be read/written until one has checked that the operation is over
 - **request** A handle that is used when checking if the operation has finished

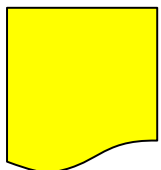
Non-blocking send

- C/C++ binding

- `int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`

- Fortran binding

- `MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)`
 - `<type>BUF(*)`
 - `INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR`



Non-blocking receive

```
int MPI_Irecv(buf, count, datatype, dest, tag,  
             comm, request )
```

- Parameters
 - Similar to MPI_Recv but has no status parameter
 - **buf** Receive buffer shall not be read/written until one has checked that the operation is over
 - **request** A handle that is used when checking if the operation has finished

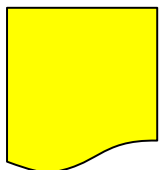
Non-blocking receive

- C/C++ binding

- `int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)`

- Fortran binding

- `MPI_IRecv(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)`
 - `<type>BUF(*)`
 - `INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR`



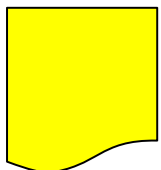
Wait for non-blocking operation

MPI_Wait(request, status)

- Parameters
 - **request** Handle of the non-blocking communication
 - **status** Status of the completed communication, see MPI_Recv
- A call to MPI_WAIT returns when the operation identified by request is complete

Wait for non-blocking operation

- C/C++ binding
 - `int MPI_Wait(MPI_Request *request, MPI_Status *status)`
- Fortran binding
 - `MPI_WAIT(REQUEST, STATUS, IERROR)`
 - `INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR`





Wait for non-blocking operations

MPI_Waitall(count, requests, statuses)

- Parameters
 - **count** Number of requests
 - **requests** Array of requests
 - **status** Array of statuses for the operations waited for
- A call to `MPI_Waitall` returns when all operations identified by the array of requests are complete

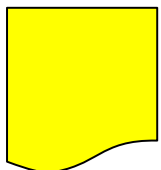
Wait for non-blocking operations

- C/C++ binding

- `int MPI_Waitall(int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses)`

- Fortran binding

- `MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)`
 - `INTEGER COUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR`

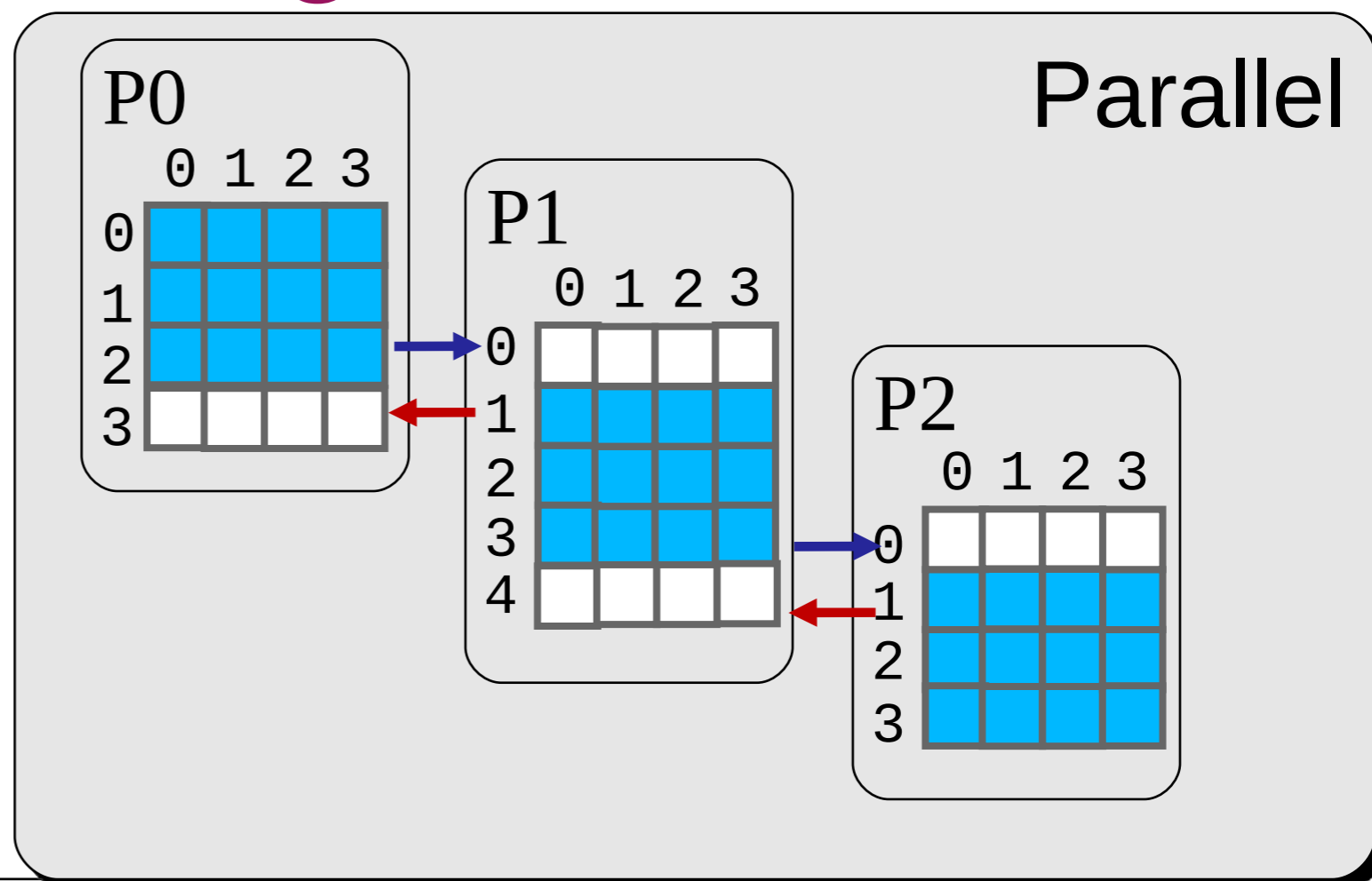


Wait operations

- Other useful routines
 - MPI_Waitany
 - MPI_Waitsome
 - MPI_Test
 - MPI_Testall
 - MPI_Testany
 - MPI_Testsome

Cs2: Non-blocking

- MPI_Isend & MPI_Irecv
- Better load balance
- Overlapping of communication & computation



Timeline

P0	ir	ir	is	is	Comp-in	wait(r)	border
P1	ir	ir	is	is	Comp-in	wait(r)	border
P2	ir	ir	is	is	Comp-in	wait(r)	border

Demonstration: Halo exchange with MPI_Isend, MPI_Irecv



- We will rewrite the algorithm using MPI_Isend, MPI_Irecv
- Let us begin...

Summary

- Non-blocking routines return immediately
- Completion of communication has to be checked separately
- No danger of deadlocks
- Enables overlapping communication and computations

Part 2 – performance considerations



Performance measurement

- In order to analyze the performance of an MPI application it is essential to be able to measure its performance
- Some options:
 - 1 Specialized tools for measuring performance
 - Powerful, but can be somewhat complex to use at first
 - Louhi:
<http://www.csc.fi/csc/kurssit/arkisto/crayworkshop09>
 - 2 Measuring the elapsed time using timer functions
 - `MPI_Wtime()`

Timing

- `MPI_Wtime()`
 - Returns a value representing the current time
 - Typical usage pattern
 1. `t=MPI_Wtime()`
 2. ...operations whose execution time is measured...
 3. `t=t-MPI_Wtime()`
- **C/C++ binding**
 - `double MPI_Wtime()`
- **Fortran binding**
 - `DOUBLE PRECISION MPI_WTIME()`

Load imbalance

- Is caused by some processes having more to do than others
 - Computation
 - Communication
 - I/O
- Can be hard to detect without performance measurement tools

MPI protocols on Louhi

- Receiver-pull protocol
 - Sender only sends once the receiver is ready
 - Default for message size >128KB on Cray XT
- Eager protocol
 - Sends immediately to temporary buffers on receiver
 - Default for small messages on Cray XT
- Some deadlocks are not activated when the eager protocol is used, can cause hard-to-track problems

Message size

- For small messages the latency dominates - effective bandwidth small
 - Aggregate messages
- Performance decreases when going above the eager limit as the sends block until the whole message has been received
 - One can attempt to use non-blocking sends to avoid

Louhi specific issues

- Pre-post receives
 - Cray XT specific issue - enables the MPI implementation to work efficiently and receive data in-place
- Overlap communication & computation
 - MPI_Irecv
 - Not efficient at overlapping
 - MPI_Isend
 - Use MPI_Send for small messages (less than eager limit)
 - Decent overlapping for large messages (over one MB)

Collectives

- One-to-all < all-to-one < all-to-all
- Use the least general routine applicable
 - i.e. do not use gatherv, alltoallv etc if you can manage with gather and alltoall
- By definition blocking communication
 - no overlapping computation and communication
- Outperforming collective routines by hacking with point-to-point routines is possible but not easy

Summary

- Parallel performance analysis is non-trivial
- Timing within the code
- Special parallel performance measurement tools
- Load balance is important for good performance