Durham University
Department of Computer Science

Core I—Introduction to HPC

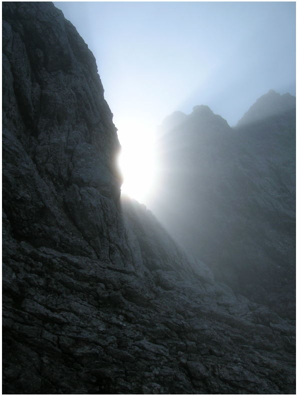
Session VIIi: Non-blocking and collective MPI

Dr. Weinzierl

Michaelmas term 2019

Outline

Durham University
Department of Computer Science



Nonblocking point to point communication
Collective operations

Buffers

Durham University
Department of Computer Science

MPI distinguishes different types of buffers:

- variables
- user-level buffers
- hardware/system buffers

MPI implementations are excellent in tuning communication, i.e. avoid copying, but we have to assume that a message runs through all buffers, then through the network, and then bottom-up through all buffers again. This means that Send and Recv are expensive operations.
Even worse, two concurrent sends might deadlock (but only for massive message counts or extremely large messages).

Sendrecv


```
int MPI_Sendrecv(
  const 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
)
```

- Shortcut for send followed by receive
- Allows MPI to optimise aggressively
- Anticipates that many applications have dedicated compute and data exchange phases

$\Rightarrow$ Does not really solve our efficiency concerns, just weaken them

Nonblocking P2P communication

Durham University
Department of Computer Science

- Non-blocking commands start with I (immediate return, e.g.)
- Non-blocking means that operation returns immediately though MPI might not have transferred data (might not even have started)
- Buffer thus is still in use and we may not overwrite it
- We explicitly have to validate whether message transfer has completed before we reuse or delete the buffer

```
Create helper variable (handle)
int a = 1;
trigger the send
do some work
check whether communication has completed
a = 2;
...
```

$\Rightarrow$ We now can overlap communication and computation.

Isend

Durham University
Department of Computer Science

```
int MPI_Send(
  const void *buffer, int count, MPI_Datatype datatype,
  int dest, int tag, MPI_Comm comm
)

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

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

- Pass additional pointer to object of type `MPI_Request`.
- Non-blocking, i.e. operation returns immediately.
- Check for send completition with `MPI_Wait` or `MPI_Test`.
- `MPI_Irecv` analogous.
- The status object is not required for the receive process, as we have to hand it over to wait or test later.

P2P communication in action


```
MPI_Request request1, request2;
MPI_Status status;
int buffer1[10]; int buffer2[10];

// Variant A
MPI_Recv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &status);
MPI_Send(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD);

// Variant B
//    MPI_Send(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD);
//    MPI_Recv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &status);

// Variant C
//    MPI_Irecv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &request1);
//    MPI_Isend(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD, &request2);
//    MPI_Wait(&request1, &status);
//    MPI_Wait(&request2, &status);
```

- Does Variant A deadlock?

P2P communication in action

Durham University
Department of Computer Science

```
MPI_Request request1, request2;
MPI_Status status;
int buffer1[10];   int buffer2[10];

// Variant A
//    MPI_Recv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &status);
//    MPI_Send(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD);

// Variant B
MPI_Send(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD);
MPI_Recv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &status);

// Variant C
//    MPI_Irecv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &request1);
//    MPI_Isend(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD, &request2);
//    MPI_Wait(&request1, &status);
//    MPI_Wait(&request2, &status);
```

- Does Variant A deadlock?
- Does Variant B deadlock?

P2P communication in action


```
MPI_Request request1, request2;
MPI_Status status;
int buffer1[10];   int buffer2[10];

// Variant A
//    MPI_Recv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &status);
//    MPI_Send(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD);

// Variant B
//    MPI_Send(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD);
//    MPI_Recv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &status);

// Variant C
MPI_Irecv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &request1);
MPI_Isend(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD, &request2);
MPI_Wait(&request1, &status);
MPI_Wait(&request2, &status);
```

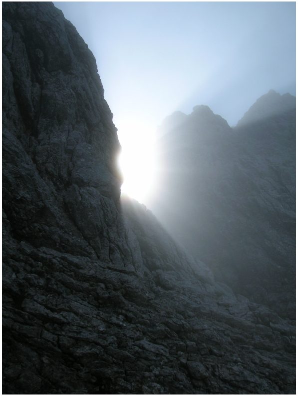
- Does Variant A deadlock?
- Does Variant B deadlock? Not for only 10 integers (if not too many messages sent before).
- Does Variant C deadlock? Is it correct? Is it fast? May we add additional operations before the first wait?

Concept of building block

- Content
  - Introduce `sendrecv`
  - Introduce concept of non-blocking communication
  - Study variants of P2P communication w.r.t. blocking and call order
- Expected Learning Outcomes
  - The student knows difference of blocking and non-blocking operations
  - The student can explain the idea of non-blocking communication
  - The student can write MPI code where communication and computation overlap

Outline

Department of Computer Science



Nonblocking point to point communication
Collective operations

Definition: collective

Durham University
Department of Computer Science

> Collective operation: A collective (MPI) operation is an operation involving many/all nodes/ranks.

- In MPI, a collective operation involves all ranks of one communicator (introduced later)
- For MPI_COMM_WORLD, a collective operation involves all ranks
- Collectives are blocking (though the newest MPI standard introduces non-blocking collectives)
- Blocking collectives always synchronise all ranks, i.e. all ranks have to enter the same collective instruction before any rank proceeds

Flavours of collective operations in MPI

Type of collective	One-to-all	All-to-one	All-to-all
Synchronisation			
Communication			
Computation			

Insert the following MPI operations into the table (MPI prefix and signature neglected):

- Barrier
- Broadcast
- Reduce
- Allgather
- Scatter
- Gather
- Allreduce

$\Rightarrow$ Looking up the signatures is homework

$\Rightarrow$ Synchronisation as discussed is simplest kind of collective operation

A (manual) collective


```
double a;

if (myrank==0) {
  for (int i=1; i<mysize; i++) {
    double tmp;
    MPI_Recv(&tmp,1,MPI_DOUBLE, ...);
    a+=tmp;
  }
}
else {
  MPI_Send(&a,1,MPI_DOUBLE,0, ...);
}
```

What type of collective operation is realised here?

A (manual) collective

Durham University
Department of Computer Science

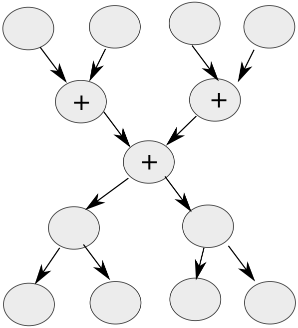
```
double a;

if (myrank==0) {
  for (int i=1; i<mysize; i++) {
    double tmp;
    MPI_Recv(&tmp,1,MPI_DOUBLE, ...);
    a+=tmp;
  }
}
else {
  MPI_Send(&a,1,MPI_DOUBLE,0, ...);
}
```

What type of collective operation is realised here?

```
double globalSum;
MPI_Reduce(&a, &globalSum, 1,
  MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Good reasons to use MPI's collective



- Simplicity of code
- Performance through specialised implementations
- Support through dedicated hardware (cf. BlueGene's three network topologies: clique, fat tree, ring)

Concept of building block

Durham University
Department of Computer Science

- Content
  - Classify different collectives from MPI
  - Study reduction in CUDA as an all-time classic of collective operations
  - Study collectives in OpenMP
- Expected Learning Outcomes
  - The student knows which type of collectives do exist (*)
  - The student can explain what collectives do (*)
  - The student can identify collective code fragments (*)
  - The student can use collectives or implement them manually