Durham University
Department of Computer Science

Core I—Introduction to HPC

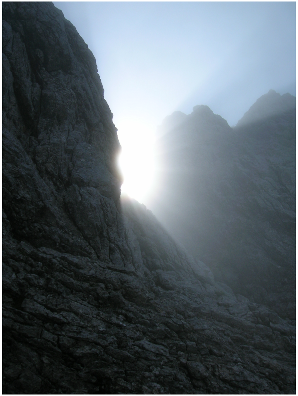
Session VII: MPI—basics and blocking P2P communication

Dr. Weinzierl

Michaelmas term 2019

Outline

Durham University
Department of Computer Science



MPI Basics
Point-to-point communication
Tags

Starting point

Durham University
Department of Computer Science

- Shared memory programming:
  - Multiple threads can access each other's data
  - Linux/OS terminology: one process with multiple tasks
- Distributed memory programming:
  - Different machines connected through a network *or*
  - one (multicore) computer running several processes, as processes do not share memory *or*
  - combinations
- De-facto standard for distributed memory programming:
  message passing (MPI = message passing interface)
- Message passing:
  - works on all/most architectural variants, i.e. is most general
    (though perhaps slower than OpenMP, e.g.)
  - requires additional coding, as we have to insert send and receive commands
  - orthogonal to other approaches, in particular to OpenMP
    (merger called MPI+X or hybrid)

Historic remarks on MPI

Durham University
Department of Computer Science

- MPI = Message Passing Interface
- Prescribes a set of functions, and there are several implementations (IBM, Intel, mpich, ...)
- Kicked-off 1992–94
- Open consortium (`www.mcs.anl.gov/mpi`)
- Mature and supported on many platforms (de-facto standard)
- Alive:
  - Extended by one-sided communication (which does not really fit to name)
  - C++ extension dropped due to lack of users
- Huge or small:
  - Around 125 functions specified
  - Most applications use only around six

A first MPI application

Durham University
Department of Computer Science

```
#include <mpi.h>

int main( int argc, char** argv ) {
  MPI_Init( &argc, &argv );
  std:::cout << "Hello world" << std::endl;
  MPI_Finalize();
  return 0;
}
```

```
mpiCC -O3 myfile.cpp
mpirun -np 10 ./a.out
```

- Use `mpiCC` which is a wrapper around your compiler
- Use `mpirun` to start application on all computers (SPMD)
- Exact usage of mpirun differs from machine to machine

A first MPI application


```
#include <mpi.h>

int main( int argc, char** argv ) {
  MPI_Init( &argc, &argv );
  std:::cout << "Hello world" << std::endl;
  MPI_Finalize();
  return 0;
}
```

- MPI functions become available through one header `mpi.h`
- MPI applications are ran in parallel (mpi processes are called **ranks** to distinguish them from OS processes)
- MPI code requires explicit initialisation and shutdown
- MPI functions always start with a prefix `MPI_` and then one uppercase letter
- MPI realises all return values via pointers
- MPI's initialisation is the first thing to do and also initialises `argc` and `argv`

MPI terminology and environment

Durham University
Department of Computer Science

Rank: MPI abstracts from processes/threads and calls each SPMD instance a *rank*. The total number of ranks is given by *size*.

```
#include <mpi.h>

int main( int argc, char** argv ) {
  MPI_Init( &argc, &argv );
  int rank,size;
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Finalize();
  return 0;
}
```

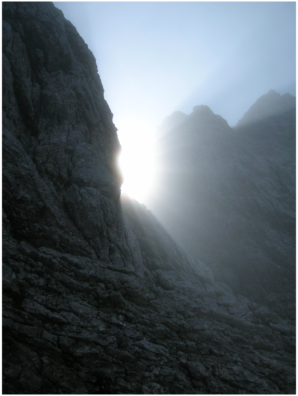
- See the name conventions and the call-by-pointer policy.
- Compare to OpenMP which offers exactly these two operations as well.
- Different to OpenMP, we will however need *ranks* all the time, as we have to specify senders and receivers.
- For the time being, rank is a continuous numbering starting from 0.

Concept of building block

Durham
University
Department of Computer Science

- Content
  - How does MPI initialise/shutdown
  - How to compile MPI codes
  - Find out local SPMD instance's rank and the global size
  - MPI conventions
- Expected Learning Outcomes
  - The student knows the framework of an mpi application, can compile it and can run it
  - The student can explain the terms rank and size
  - The student can identify MPI conventions at hands of given source codes

Outline



MPI Basics
Point-to-point communication
Tags

Send


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

- `buffer` is a pointer to the piece of data you want to send away.
- `count` is the numer of items
- `datatype` . . . self-explaining
- `dest` is the rank (integer) of the destination node
- `comm` is the so-called communicator (always `MPI_COMM_WORLD` for the time being)
- Result is an error code, i.e. 0 if successful (UNIX convention)

Blocking: MPI_Send is called `blocking` as it terminates as soon as you can reuse the buffer, i.e. assign a new value to it, without an impact on MPI.

Receive


```
int MPI_Recv(
  void *buffer, int count, MPI_Datatype datatype,
  int source, int tag, MPI_Comm comm,
  MPI_Status *status
)
```

- `buffer` is a pointer to the variable into which the received data shall be stored.
- `count` is the numer of items
- `datatype` ...self-explaining
- `dest` is the rank (integer) of the source node (may be `MPI_ANY`)
- `comm` is the so-called communicator (always `MPI_COMM_WORLD` for the time being)
- `status` is a pointer to an instance of `MPI_Status` and holds meta information
- Result is an error code, i.e. 0 if successful (UNIX convention)

Blocking: MPI_Recv is called `blocking` as it terminates as soon as you can read the buffer, i.e. MPI has written the whole message into this variable.

Blocking communication

Blocking: MPI_Send is called `blocking` as it terminates as soon as you can reuse the buffer, i.e. assign a new value to it, without an impact on MPI.

Blocking: MPI_Recv is called `blocking` as it terminates as soon as you can read the buffer, i.e. MPI has written the whole message into this variable.

- If a blocking operation returns, it does **not** mean that the corresponding message has been received.
- Blocking and asynchronous or synchronous execution have nothing to do with each other though a blocking receive never returns before the sender has sent out its data.
- If a blocking send returns, the data must might have been copied to the local network chip.
- The term blocking just refers to the safety of the local variable.
- With blocking sends, you never have a guarantee that the data has been received, i.e. blocking sends are not *synchronised*.

First code

What are the values of a and b on rank 2?

```
if (rank==0) {
  int a=0;
  MPI_Send(&a, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
  a=1;
  MPI_Send(&a, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
}
if (rank==2) {
  MPI_Status status;
  int a;
  MPI_Recv(&a, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  int b;
  MPI_Recv(&b, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
}
```

- MPI messages *from one rank* never overtake.
- Why is this called SPMD?

MPI Datatypes


```
MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
```

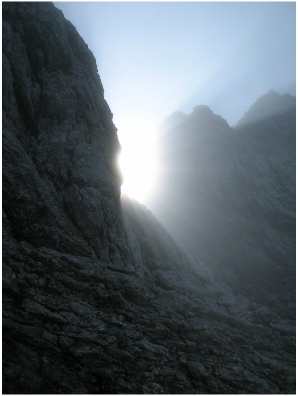
- There are more data types predefined.
- However, I've never used others than these.
- Note that there is no bool (C++) before MPI-2.
- In theory, heterogeneous hardware supported.
- Support for user-defined data types and padded arrays.

Concept of building block

Durham University
Department of Computer Science

- Content
  - Study MPI send and receive
  - Discuss term blocking
  - Introduce predefined data types
  - Study a simple MPI program
- Expected Learning Outcomes
  - The student knows signature and semantics of MPI's send and receive
  - The student knows the predefined MPI data types
  - The student can explain the term blocking
  - The student can write down a simple correct MPI program (incl. the initialisation and the shutdown)
- Material & further reading
  - Gropp et al.: Tutorial on MPI: The Message-Passing Interface

Outline



MPI Basics
Point-to-point communication
Tags

MPI tags

Durham University
Department of Computer Science

Tag: A **tag** is a meta attribute of the message when you send it away. The receive command can filter w.r.t. tags.

Message arrival: Two MPI messages *with the same tag may not overtake*.

- With tags, we can make messages overtake each other.
- Tags are typically used to distinguish messages with different semantics.
- Tags are arbitrary positive integers. There is no need to explicitly register them.
- Extreme scale: Too many tags might mess up MPI implementation.
- For sends, real tag is mandatory. Receives may use `MPI_ANY_Tag` (wildcard).

Example 1/2

Durham University
Department of Computer Science

The following snippet shall run on a rank p0:

```
int a=1;
int b=2;
int c=3;
MPI_Send(&a,1,MPI_INT,p1,0,MPI_COMM_WORLD);
MPI_Send(&b,1,MPI_INT,p1,0,MPI_COMM_WORLD);
MPI_Send(&c,1,MPI_INT,p1,1,MPI_COMM_WORLD);
```

The following snippet shall run on a rank p1:

```
int u;
int v;
int w;
MPI_Recv(&u,1,MPI_INT,p0,0,MPI_COMM_WORLD);
MPI_Recv(&v,1,MPI_INT,p0,0,MPI_COMM_WORLD);
MPI_Recv(&w,1,MPI_INT,p0,0,MPI_COMM_WORLD);
```

What is the value of u,v,w on rank p1?

Example 2/2

The following snippet shall run on a rank `p0`:

```
int a=1;
int b=2;
int c=3;
MPI_Send(&a,1,MPI_INT,p1,0,MPI_COMM_WORLD);
MPI_Send(&b,1,MPI_INT,p1,0,MPI_COMM_WORLD);
MPI_Send(&c,1,MPI_INT,p1,1,MPI_COMM_WORLD);
```

The following snippet shall run on a rank `p1`:

```
int u;
int v;
int w;
MPI_Recv(&u,1,MPI_INT,p0,0,MPI_COMM_WORLD);
MPI_Recv(&v,1,MPI_INT,p0,1,MPI_COMM_WORLD);
MPI_Recv(&w,1,MPI_INT,p0,0,MPI_COMM_WORLD);
```

What is the value of u,v,w on rank `p1`?

Excursus/addendum: Buffered sends

Durham University
Department of Computer Science

```
int  bufsize;
char *buf = malloc(bufsize);
MPI_Buffer_attach( buf, bufsize );
...
MPI_Bsend( ... same as MPI_Send ... );
...
MPI_Buffer_detach( &buf, &bufsize );
```

- If you use many tags, many blocking commands, and so forth, you stress your system buffers (variables, MPI layer, hardware)
- This might lead to deadlocks though your code semantically is correct
- MPI provides a send routine that buffers explicitly: MPI_Bsend
- MPI_Bsend makes use of a user-provided buffer to save any messages that can not be immediately sent.
- Buffers explicitly have to be added to MPI and removed at program termination.
- The MPI_Buffer_detach call does not complete until all messages are sent.

Concept of building block

- Content
  - Introduce definition of a tag
  - Make definition of not-overtaking explicit and study usage
  - Introduce buffered sends
- Expected Learning Outcomes
  - The student knows definition of tags
  - The student can explain their semantics w.r.t. message arrival
  - The student can use tags in MPI statements