Message Passing Interface

Lecture 6

Stanisław Gepner

27th April, 2023

Introduction

We will now step up to the real HPC world, where not merely cores, limited to one computer but a number of separate computers with all their resources, connected by network can be used. This approach (opposed to SMP) has a benefit of virtually unlimited resources. For concurrency control we will employ MPI - Message Passing Interface standard wrapped in one of many of its implementations. Luckily, the standard defines the interface itself, and we need only to worry about our code and making it parallel.

For now we forget about threads, with MPI we will be considering only separate processes that communicate via MPI standardized interface.

MPI stands for Message Passing Interface. It is a standard for handling process creation and communication over a single or multiple mashines connected by the network.

There is a number of implementations.

MPI first program

Initialisation and Finalisation

Let us start with initialisation: MPI_Init(NULL, NULL)

rank, and the number of processes.

```
#include <mpi.h>

MPI_Init(NULL, NULL);

int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

MPI_Finalize();
```

Every MPI program must end with MPI_Finalize();

Program build with MPI need an MPI agent to start and to assign processes to cores and processors.

```
mpirun -n N mpi_program_executable
```

Simple Send and Recive - Deadlock

Blocking Send - Recive

```c
#include <mpi.h>
int MPI_Send( void * msg, int count,
    MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm );
int MPI_Recv( void * msg, int count,
    MPI_Datatype datatype, int source,
    int tag, MPI_Comm comm,
    MPI_Status * status );
```

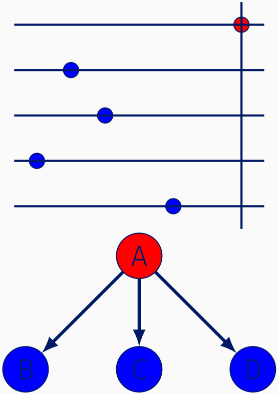
Send msg from source do dest, and recive at dest from source in a Blocking manner.

Deadlock is possible.

Collective communication - all together now

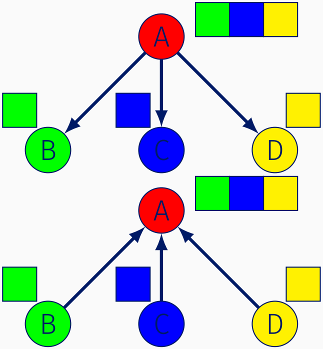
Collective communication allows to pass messages in between all processes. There is a number of possibilities, either one process communicates to all, or all comunicate to one, or it is an all-to-all message passing.

```
#include <mpi.h>
int MPI_Barrier( MPI_Comm comm );

int MPI_Bcast( void * buffer, int count,
               MPI_Datatype datatype,
               int root, MPI_Comm comm );
```



Barrier is used to synchronise processes and Brodcast allows to pass a messae from one to all.

```
#include <mpi.h>
int MPI_Scatter
    (const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf,
    int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm);

int MPI_Gather
    (const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf,
    int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm);
```



Scatter send a portion of the sendbuf from one to all, while Gather collects from all to one.

Collective communication - reduction

Reduce and All Reduce

```c
#include <mpi.h>
int MPI_Reduce( void * sendbuf, void * recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm );
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Applies one of reduction operations to distributed data, depending on the verison one or all processes gets the result.

- MPI_MAX maximum
- MPI_MIN minimum
- MPI_SUM sum
- MPI_PROD product
- MPI_MAXLOC max value and location
- MPI_MINLOC min value and location

Asynchronous communication

The standard send and recive are blocking calls, meaning control is blocked until the send / receive operation is complete. The upside is that it is guaranteed that the send / receive buffer is safe for writing again or reading from. But on the other hand deadlock is possible.

An alternative is the nonblocking comunication. Here we state the ntention to send / receive communication and control is not locked, but program keeps on executing. The downside is that additional care must be taken before overriding or reading from the send / receive buffer.

Isend Irecive and Wait

```
#include <mpi.h>

MPI_Request request;
MPI_Status  status;

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

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