Computação Paralela / Computação Avançada

2022–2023 1st semester

Helmut Wolters
helmut@coimbra.lip.pt

LIP / Universidade de Coimbra

cap5 — 2022-11-09

Message Passing Interface — MPI

Detecting deadlock problems

Whenever we use messages through MPI between processes, there is the danger to misconfigure the interaction.

Example:

- The master process waits for the messages with the results from the slave processes
- One slave process hangs or terminates with an error, and never sends the message
- Thus the master process will hang waiting for the message and never return

This can be due to

- a wrong design of the message interactions, or
- an unexpected execution error in one of the processes

Message Passing Interface — MPI

Detecting deadlock problems

Thus, during the development and testing of a parallel program, it is useful to implement a verbose logging of all messages sent and expected to be reveived, so that in a debug session we can verify where an expected message was never received.

This leads to a huge noise of log messages, but without all of them we have no chance to track down the problem.

But for the production run of the program, it should be easy to remove these logs.

See `dice8.py` for a suggestion of implementing such a debug system .

Message Passing Interface — MPI

Collective communication using broadcasts

- We often want to send the same message from one process to all the other (e.g. master to all slaves)
- We can repeat to send the same message using a loop
- MPI offers a better method for this, called *broadcast*.
- Advantages:
  - it simplifies considerably the code
  - it allows MPI to use more efficient communication methods to implement a broadcast

Message Passing Interface — MPI

Collective communication using broadcasts

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

v = 100 * (rank + 1)

print("rank %d" % rank + " v = %d " % v)
msg = comm.bcast(v, root=0)
print("rank", rank, "msg =", msg)
```

Note that there is no fork necessary to distinguish the master process from the slaves. The process with rank specified by "root" sends the message to all the others, and all processes receive it, including the root itself.

Message Passing Interface — MPI

Collective communication using scatter

The scatter functionality is very similar to a broadcast but has one major difference.

While a broadcast sends the same data to all listening processes, scatter can send the chunks of data in a list to different processes.

This comes in handy if the master process needs to distribute different parts of the work to each of the slaves.

Message Passing Interface — MPI

Collective communication using scatter

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
   data = [100, 200, 300, 400]
else:
   data = None

v = comm.scatter(data, root=0)
print("rank", rank, "received v =", v )
```

Message Passing Interface — MPI

Collective communication using gather

The gather function performs the inverse of the scatter functionality. In this case, all processes send data to a root process that collects the data received.

This comes in handy for the master to collect the results of all of the slave processes.

Message Passing Interface — MPI

Collective communication using gather

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
data = (rank+1)**2
print("rank", rank, "creates data =", data )

msg = comm.gather(data, root=0)

print("rank", rank, "gathers msg =", msg )
```