Parallel Programming Exercises

Andreas Hollmann

Technische Universität München

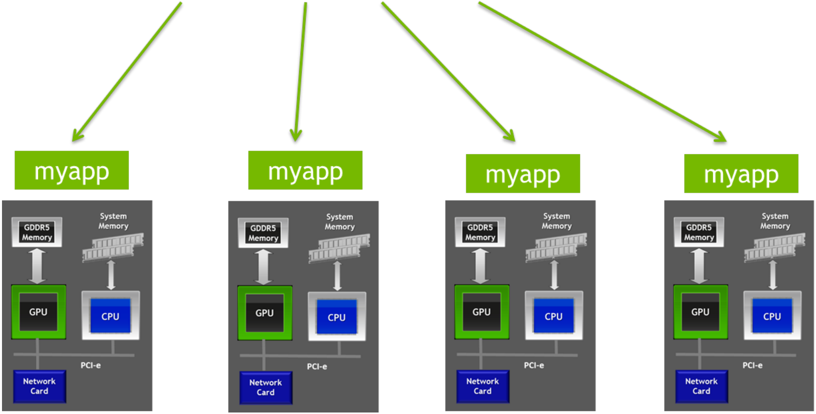
June 20, 2016

MPI: Scaling beyond Shared Memory

- Pthreads and OpenMP work only on shared memory
- Big problems/simulations don't fit into a single shared memory node or take too long to compute
- Message Passing Interface (MPI) allows scaling across distributed memory
- Used on all Supercomputers

MPI: Execution

```
mpirun -np 4 ./myapp <args>
```



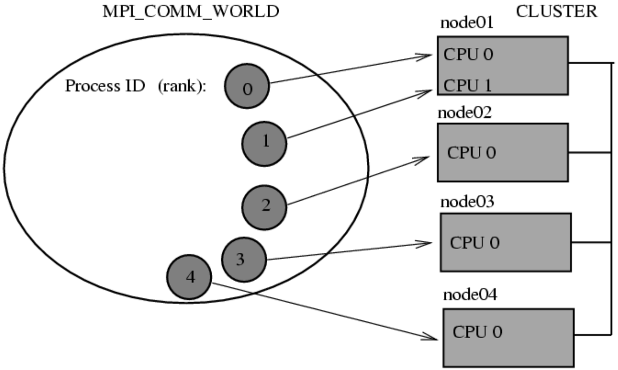
MPI: Message Passing Interface (1)

- Library + Tools (compiler wrapper, documentation, deamon)
- Enables writing applications on distributed memory and shared memory systems.
- Communications is done by sending messages.
- Single Program Multiple Data (SPMD) programming model
- Single Program(source), is started as (multiple) processes on local or remote machines. Each process works on local data.

MPI: Message Passing Interface (2)

- Two types of operations: point-to-point and collective
- Each process in a communicator is identified by its rank (id)
- Work distribution is done by using rank information.
- All data is private. If data has be accessed by an another process, it has to be send to this process.

MPI: Overview



MPI: Hello world!

```c
#include <mpi.h>
#include <stdio.h>

int main (int argc, char* argv[])
{
  int rank, size;

  MPI_Init(&argc, &argv); /* starts MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* process id */
  MPI_Comm_size(MPI_COMM_WORLD, &size); /* number processes */
  printf( "Hello world from process %d of %d\n", rank, size );
  MPI_Finalize();
  return 0;
}
```

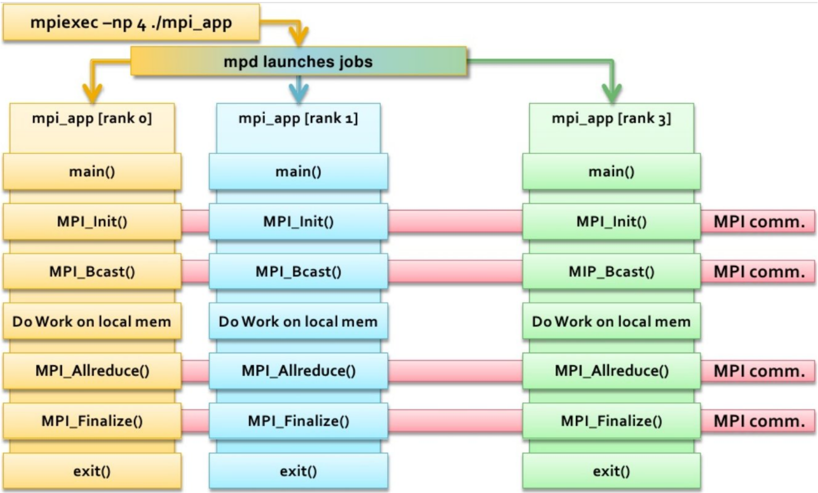
MPI: Compilation & Execution

```
$ mpicc mpi_hello.c -o hello
$ mpirun -np 2 ./hello
Hello world from process 0 of 2
Hello world from process 1 of 2
```

MPI: Message Passing Interface (3)

- MPI runtime handles the startup of all processes and takes care about the enumeration of the processes (ranks).
- Distribution of processes to machines can be configured, but this is not part of the exercise.
- You will work locally with MPI, but there's no difference to working on a remote machine, except of performance.

MPI: Execution



MPI Debugging

- Debugging is hard with MPI, worse than OpenMP or Pthreads, because of multiple processes. But it is more explicit.
- This makes writing MPI applications time consuming.
- Debugging can be done by using printf(). MPI redirects the output to your local terminal.
- Debugger: Commercial MPI debuggers (totalview) and a plugin for Eclipse: Parallel Tools Platform (PTP)

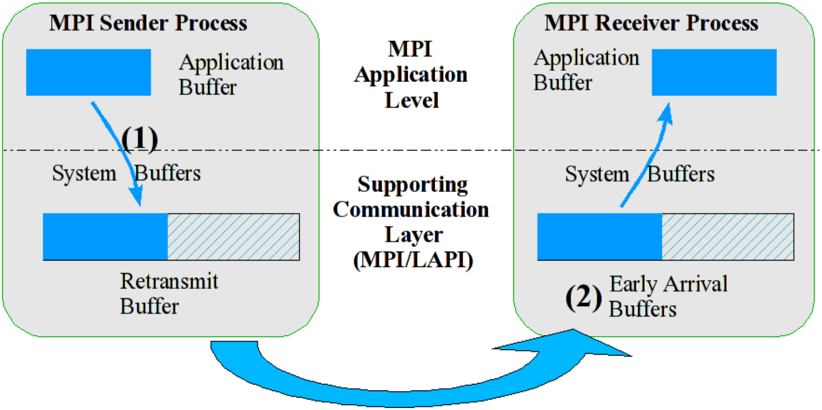
MPI: Installation Ubuntu

- $ sudo apt-get install libcr-dev libopenmpi-dev openmpi-bin openmpi-doc
- OR
- $ sudo apt-get install libcr-dev mpich2 mpich2-doc
- Only install one of these two MPI libraries

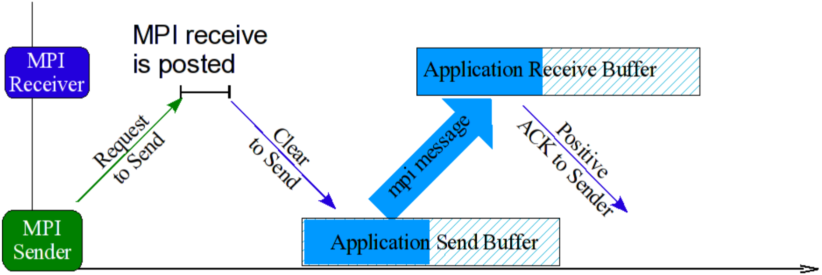
MPI: Message Passing Interface

- Most MPI calls are blocking, e. g. MPI_Send, MPI_Recv
- This is important to know, to avoid deadlocks!
- Send doesn't block until message is received, but only until data is copied into internal buffer if there is enough space.
- If the message does not fit into the buffer, a different protocol is used to send the message: rendezvous-protocol. Both MPI_Send and the according MPI_Recv have to be called to avoid a deadlock.

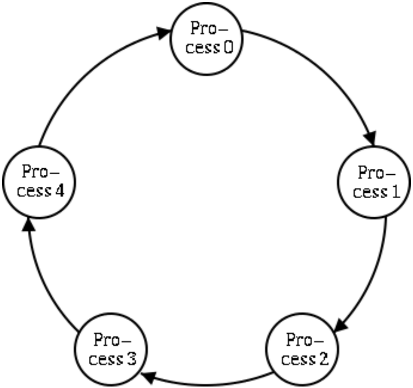
MPI: Data Flow



MPI: Rendezvous-Protocol



MPI: Right-Shift Example



MPI: Does this always work? (1)

```
int main (int argc, char* argv[])
{
  int rank, size, tmp[10000];

  MPI_Init(&argc, &argv); /* starts MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* process id */
  MPI_Comm_size(MPI_COMM_WORLD, &size); /* number processes */

  MPI_Send(tmp, atoi(argv[1]), MPI_INT, mod(rank+1,size), 0,
                MPI_COMM_WORLD);
  MPI_Recv(tmp, atoi(argv[1]), MPI_INT, mod(rank-1,size), 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  MPI_Finalize();
  return 0;
}
```

MPI: Does this always work? (2)

```
int main (int argc, char* argv[])
{
  int rank, size, tmp[10000];
  ...

  if(rank == 0)
  { MPI_Recv(tmp, atoi(argv[1]), MPI_INT, mod(rank-1,size), 0,
      MPI_COMM_WORLD, MPI_STATUS_IGNORE); }
      MPI_Send(tmp, atoi(argv[1]), MPI_INT, mod(rank+1,size), 0,
      MPI_COMM_WORLD); }
  else
  { MPI_Send(tmp, atoi(argv[1]), MPI_INT, mod(rank+1,size), 0,
      MPI_COMM_WORLD);
      MPI_Recv(tmp, atoi(argv[1]), MPI_INT, mod(rank-1,size), 0,
      MPI_COMM_WORLD, MPI_STATUS_IGNORE); }

  MPI_Finalize();
  return 0;
}
```

MPI: Does this always work? (3)

```
int main (int argc, char* argv[])
{
  int rank, size, tmp;

  MPI_Init(&argc, &argv); /* starts MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* process id */
  MPI_Comm_size(MPI_COMM_WORLD, &size); /* number processes */

  MPI_Sendrecv(tmp, atoi(argv[1]), MPI_INT, mod(rank+1,size),
               tmp, atoi(argv[1]), MPI_INT, mod(rank-1,size),
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  MPI_Finalize();
  return 0;
}
```

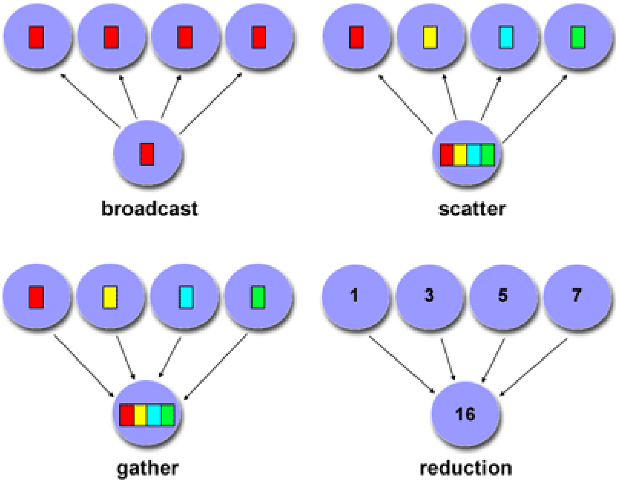
MPI_Sendrecv()

- Send and Recv are called as if by two independent threads!
- No deadlock that can be caused by the order of execution of Send and Recv.
- Deadlocks can still occur, if Send and Recv signatures don’t match.
- MPI tag and source has to match.
- MPI status can be ignored by using MPI_STATUS_IGNORE.
- MPI_ANY_TAG and MPI_ANY_SOURCE can be used if tag and source are of no interest.
- Use the source identifier to make sure that you send and receive from the right process.

MPI: Important Functions

- MPI_Send(), MPI_Recv(), MPI_Sendrecv()
- asynchronous versions: MPI_Isend(), MPI_Irecv(), MPI_Wait(), MPI_Test()
- Collective Operations:
  - MPI_Broadcast()
  - MPI_Reduce()
  - MPI_Scatter() – MPI_Scatterv() only for assignment
  - MPI_Gather() – MPI_Gatherv() only for assignment
  - MPI_Barrier()

MPI: Important Collectives



MPI_Scatterv() and MPI_Gatherv()

- Like MPI_Scatter() and MPI_Gather() but can work on sparse / unregular data
- Two additional parameters, pointer to arrays (one element per rank) that hold the number of elements and the index of elements to scatter or gather.
- Take a look at the online documentation for further details.

Assignment: Reversing a (huge) char buffer with MPI

- Input e.g: This is a simple string that should be printed in reverse order
- Output: redro esrever ni detnirp eb dluohs taht gnirts elpmis a si sihT
- Has to work with any number of processes (np < number of chars)
- You can reuse reverse function for local computation

Assignment: Reversing a (huge) char buffer with MPI

- 3 steps necessary to parallelize the application.
  - Distribute array from rank 0 to all ranks using MPI_Scatterv().
  - Call provided reverse function on the local part of the array.
  - Send local part of the array back to Rank 0 and store it directly at the right position.
- Implement scatterv first and make sure that it's working correctly!!! You can use the provided print function to print the char buffer.
- Use only following MPI Routines: Scatterv(), Send(), Recv() and nothing else!!
- MPI template of the assignment will be provided.

Assignment: Reversing a (huge) char buffer with MPI