Introduction to parallel computing with MPI and Python

Robert Klöfkorn

2021

Question?

Who has used a parallel computer?

Question?

Who has used a parallel computer?

```
blade ~ $ lscpu -e
CPU NODE SOCKET CORE L1d:L1i:L2:L3 ONLINE    MAXMHZ   MINMHZ
  0    0      0    0 0:0:0:0          yes 4500.0000 800.0000
  1    0      0    1 1:1:1:0          yes 4500.0000 800.0000
  2    0      0    2 2:2:2:0          yes 4500.0000 800.0000
  3    0      0    3 3:3:3:0          yes 4500.0000 800.0000
  4    0      0    4 4:4:4:0          yes 4500.0000 800.0000
  5    0      0    5 5:5:5:0          yes 4500.0000 800.0000
  6    0      0    0 0:0:0:0          yes 4500.0000 800.0000
  7    0      0    1 1:1:1:0          yes 4500.0000 800.0000
  8    0      0    2 2:2:2:0          yes 4500.0000 800.0000
  9    0      0    3 3:3:3:0          yes 4500.0000 800.0000
 10    0      0    4 4:4:4:0          yes 4500.0000 800.0000
 11    0      0    5 5:5:5:0          yes 4500.0000 800.0000
blade ~ $
```

Why Parallelization?

Moore's law is the observation that the number of transistors in a dense integrated circuit (IC) doubles about every two years. Taking into account various factors that meant that computer chip performance would roughly double every 18 months.

Why Parallelization?

Moore's law is the observation that the number of transistors in a dense integrated circuit (IC) doubles about every two years. Taking into account various factors that meant that computer chip performance would roughly double every 18 months.

Herb Sutter. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.* Dr. Dobb's Journal, 30(3), March 2005.

Why Parallelization?

Moore's law is the observation that the number of transistors in a dense integrated circuit (IC) doubles about every two years. Taking into account various factors that meant that computer chip performance would roughly double every 18 months.

Herb Sutter. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.* Dr. Dobb's Journal, 30(3), March 2005.

- Speedup numerical algorithm by using more processors
- Problem size may exceed a single processors memory
- ...

Why Parallelization?

Moore's law is the observation that the number of transistors in a dense integrated circuit (IC) doubles about every two years. Taking into account various factors that meant that computer chip performance would roughly double every 18 months.

Herb Sutter. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.* Dr. Dobb's Journal, 30(3), March 2005.

- Speedup numerical algorithm by using more processors
  Example: Spinning up a climate model takes 130 days on a super computer. Here, better parallelization can drive down the waiting time and thus cost of running the model.
- Problem size may exceed a single processors memory
- ...

Why Parallelization?

Moore's law is the observation that the number of transistors in a dense integrated circuit (IC) doubles about every two years. Taking into account various factors that meant that computer chip performance would roughly double every 18 months.

Herb Sutter. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.* Dr. Dobb's Journal, 30(3), March 2005.

- Speedup numerical algorithm by using more processors
  Example: Spinning up a climate model takes 130 days on a super computer. Here, better parallelization can drive down the waiting time and thus cost of running the model.
- Problem size may exceed a single processors memory

- ...

Why Parallelization?

Moore's law is the observation that the number of transistors in a dense integrated circuit (IC) doubles about every two years. Taking into account various factors that meant that computer chip performance would roughly double every 18 months.

Herb Sutter. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.* Dr. Dobb's Journal, 30(3), March 2005.

- Speedup numerical algorithm by using more processors
  Example: Spinning up a climate model takes 130 days on a super computer. Here, better parallelization can drive down the waiting time and thus cost of running the model.
- Problem size may exceed a single processors memory
  Example: Climate models easily have 10^{10} number of unknowns, a single computer with 8GB memory can at most store $N \approx 10^9$ floating point numbers
- ...

Why Parallelization?

Moore's law is the observation that the number of transistors in a dense integrated circuit (IC) doubles about every two years. Taking into account various factors that meant that computer chip performance would roughly double every 18 months.

Herb Sutter. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.* Dr. Dobb's Journal, 30(3), March 2005.

- Speedup numerical algorithm by using more processors
  Example: Spinning up a climate model takes 130 days on a super computer. Here, better parallelization can drive down the waiting time and thus cost of running the model.
- Problem size may exceed a single processors memory
  Example: Climate models easily have 10^{10} number of unknowns, a single computer with 8GB memory can at most store $N \approx 10^9$ floating point numbers
- ...

MIMD – multiple instruction, multiple data

Machines using MIMD have a number of processors that function asynchronously and independently. At any time, different processors may be executing different instructions on different pieces of data.

- Shared Memory
- Distributed Memory

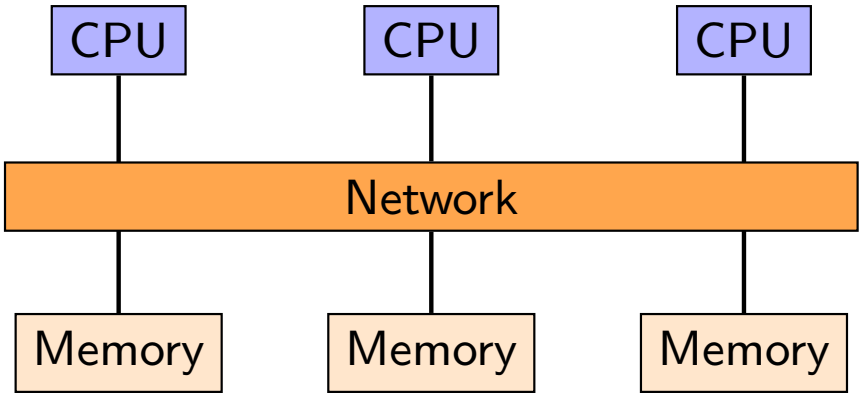
Programming paradigms: Shared memory

Shared memory

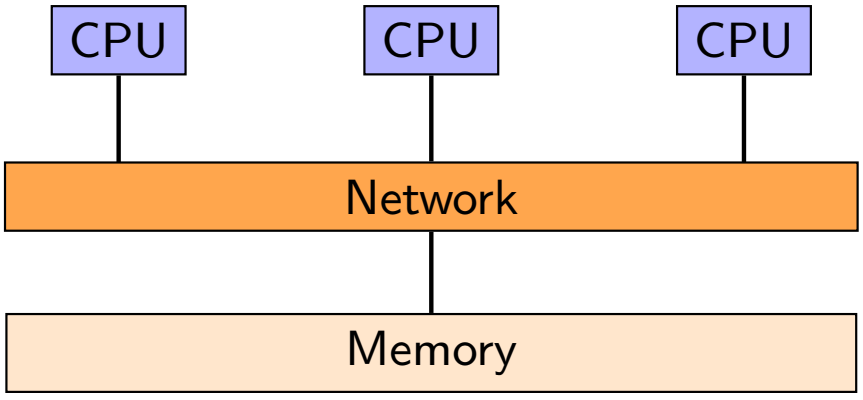
- each CPU has it's own processes and these can share memory
- necessity to ensure correct memory access, i.e. order access to same memory address (race condition)
- easy to program at first (for small code sections), race conditions can be a real nightmare in larger codes

Today's commonly used is for example **Open Multi Processing (OpenMP)** or Intel's **Thread Building Blocks (TBB)**.

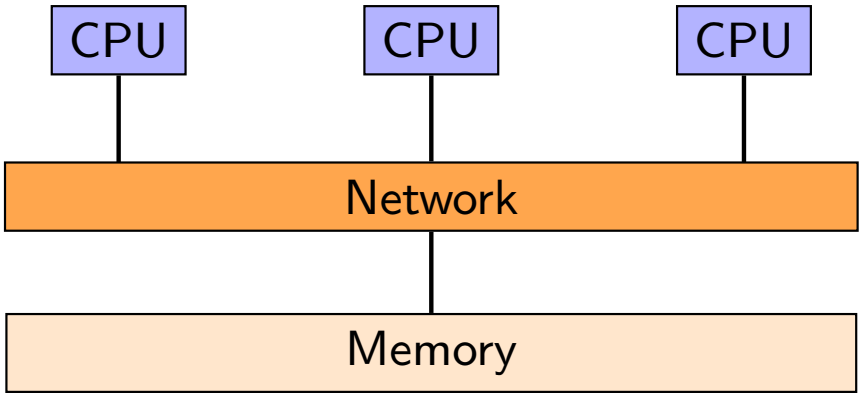
Shared Memory MIMD – Example 1



Shared Memory MIMD – Example 2



Shared Memory MIMD – Example 2



- Usually limited to smaller number of CPUs (< 1000)

Programming paradigms: Distributed memory

Distributed memory

- each CPU has it's own process but also it's own memory
- processes communicate by passing messages to each other (Message Passing)
- synchronization

Programming paradigms: Distributed memory

Distributed memory

- each CPU has it's own process but also it's own memory
- processes communicate by passing messages to each other (Message Passing)
- synchronization

Today's standard is the so-called **Message Passing Interface (MPI)**.
Variants of MPI implementations

- MPICH, first implementation 1994, many forks exists
- OpenMPI, open source implementation to prevent forking
  OpenMPI != OpenMP
- MVAPICH, Intel MPI, MS MPI, and many more

Issues to keep in mind

- Communication is much slower than Computation! Thus avoid communication as much as possible (communication avoiding algorithms)
- Memory per core will decrease
- Degree of parallelism will increase
- Memory access is also slow compared to computations (FLOPs are for free)

Issues to keep in mind

- Communication is much slower than Computation! Thus avoid communication as much as possible (communication avoiding algorithms)
- Memory per core will decrease
- Degree of parallelism will increase
- Memory access is also slow compared to computations (FLOPs are for free)

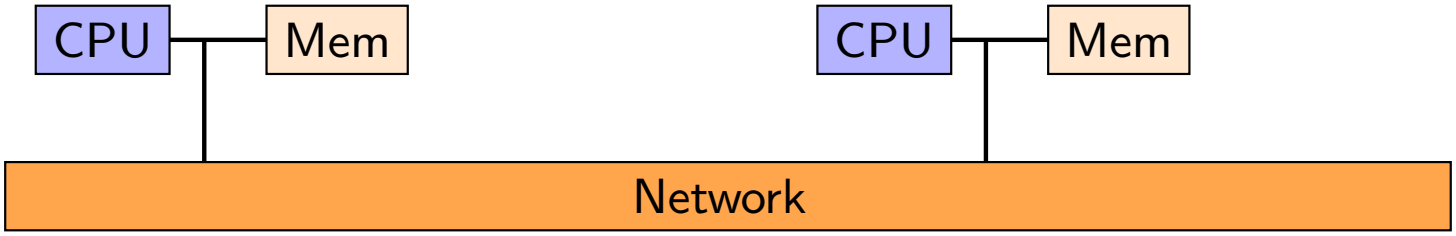
Issues to keep in mind

- Communication is much slower than Computation! Thus avoid communication as much as possible (communication avoiding algorithms)
- Memory per core will decrease
- Degree of parallelism will increase
- Memory access is also slow compared to computations (FLOPs are for free)

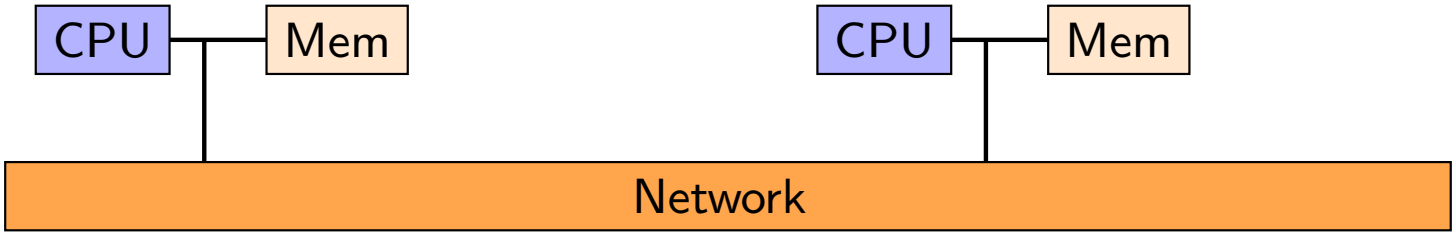
Issues to keep in mind

- Communication is much slower than Computation! Thus avoid communication as much as possible (communication avoiding algorithms)
- Memory per core will decrease
- Degree of parallelism will increase
- Memory access is also slow compared to computations (FLOPs are for free)

Distributed Memory architecture

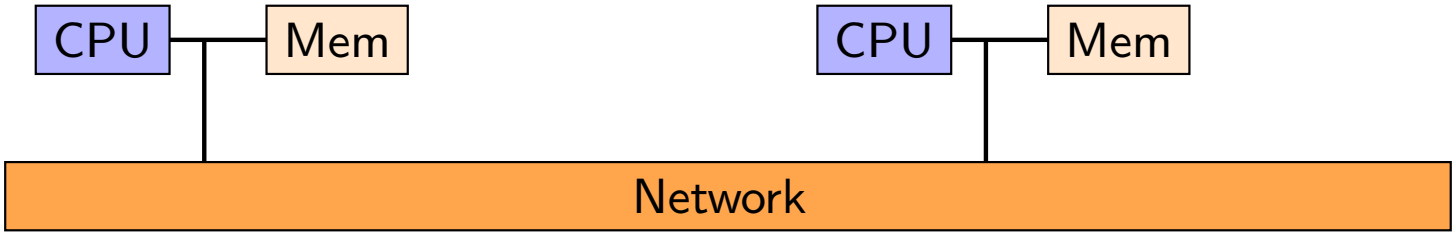


Distributed Memory architecture



- A CPU–Memory pair is called a (work) node.
- Common model is the SPMD (single program multiple data)
- A node could be a SIMD (single instruction multiple data) or shared memory MIMD node.

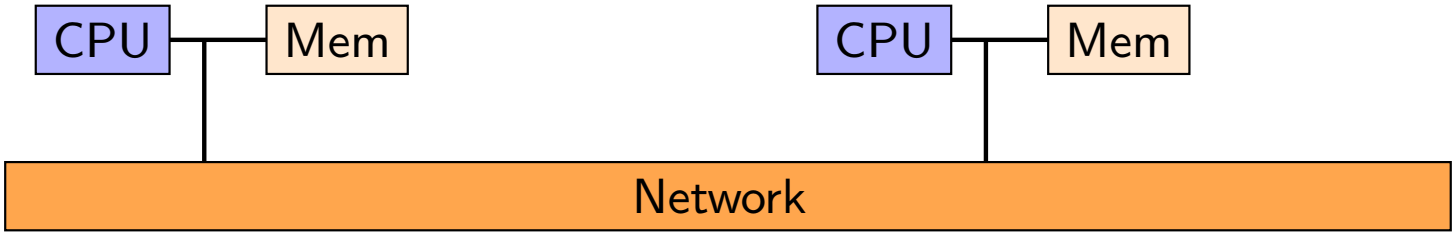
Distributed Memory architecture



- A CPU–Memory pair is called a (work) node.
- Common model is the SPMD (single program multiple data)
- A node could be a SIMD (single instruction multiple data) or shared memory MIMD node.

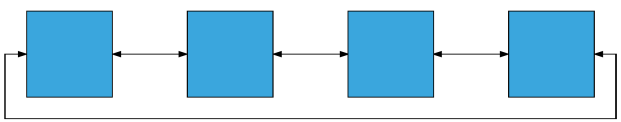
How does the network look like?

Distributed Memory architecture



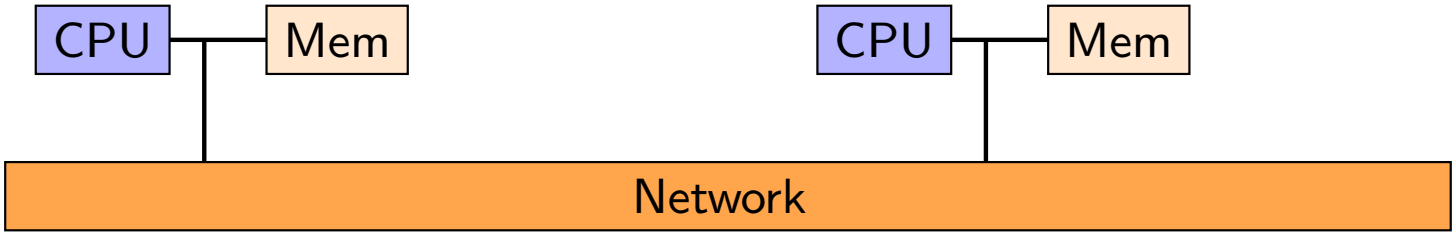
- A CPU–Memory pair is called a (work) node.
- Common model is the SPMD (single program multiple data)
- A node could be a SIMD (single instruction multiple data) or shared memory MIMD node.

Network topology

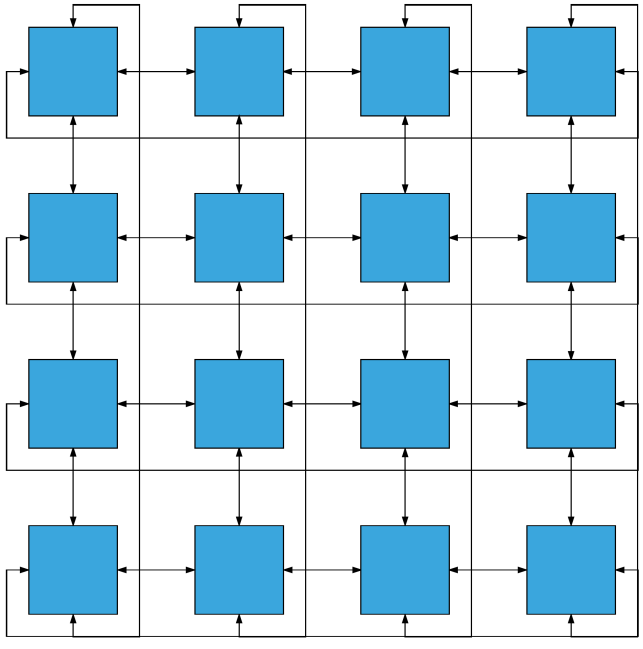


1d torus (from Wikipedia)

Distributed Memory architecture

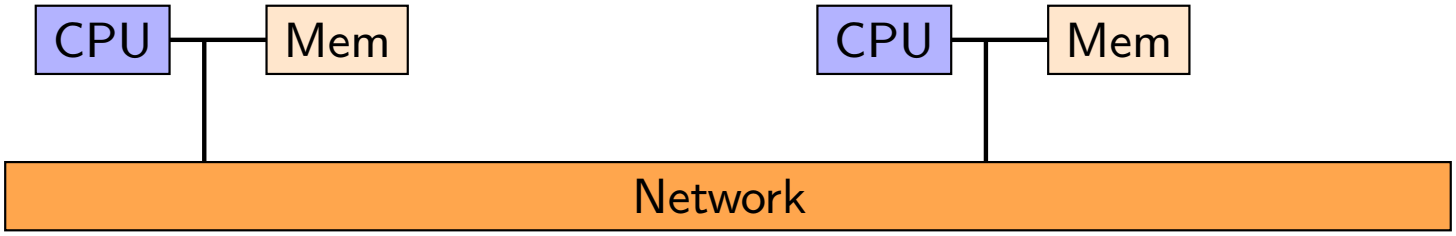


Network topology

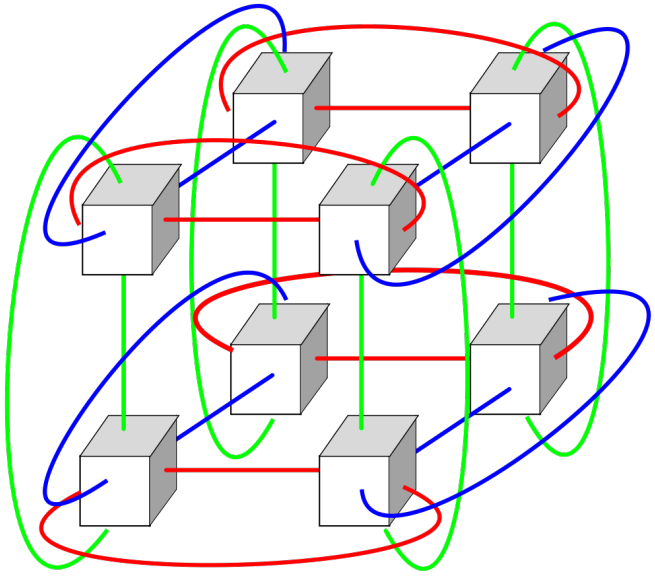


2d torus (from Wikipedia)

Distributed Memory architecture

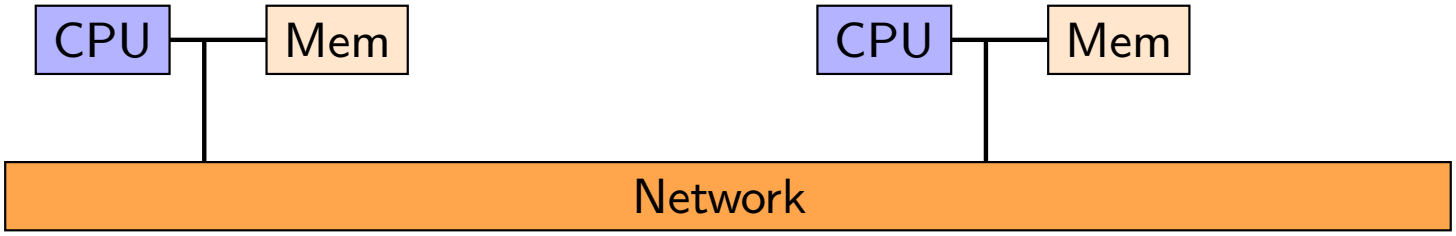


Network topology

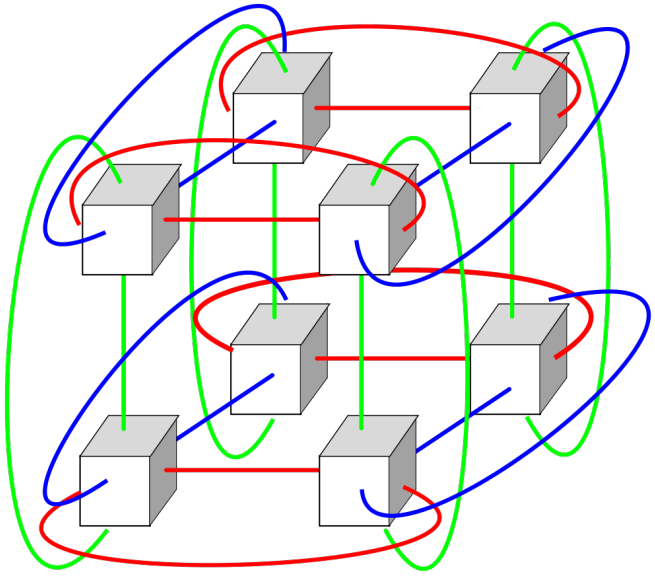


3d torus (from Wikipedia)

Distributed Memory architecture



Network topology



3d torus (from Wikipedia)

Also HyperCube networks and mesh networks.

The mpi4py module

- Python interface to MPI
- Based on MPI-2 C/C++ bindings
- Almost all MPI calls supported
- Popular on Linux clusters and in the SciPy community
- Operations are primarily methods on communicator objects
- Supports communication of pickled (serializable) Python objects
- Optimized communication of NumPy arrays
- API docs: `https://mpi4py.readthedocs.io/en/stable/`

The mpi4py module

- Python interface to MPI
- Based on MPI-2 C/C++ bindings
- Almost all MPI calls supported
- Popular on Linux clusters and in the SciPy community
- Operations are primarily methods on communicator objects
- Supports communication of pickled (serializable) Python objects
- Optimized communication of NumPy arrays
- API docs: `https://mpi4py.readthedocs.io/en/stable/`

The mpi4py module

- Python interface to MPI
- Based on MPI-2 C/C++ bindings
- Almost all MPI calls supported
- Popular on Linux clusters and in the SciPy community
- Operations are primarily methods on communicator objects
- Supports communication of pickled (serializable) Python objects
- Optimized communication of NumPy arrays
- API docs: `https://mpi4py.readthedocs.io/en/stable/`

The mpi4py module

- Python interface to MPI
- Based on MPI-2 C/C++ bindings
- Almost all MPI calls supported
- Popular on Linux clusters and in the SciPy community
- Operations are primarily methods on communicator objects
- Supports communication of pickled (serializable) Python objects
- Optimized communication of NumPy arrays
- API docs: `https://mpi4py.readthedocs.io/en/stable/`

The mpi4py module

- Python interface to MPI
- Based on MPI-2 C/C++ bindings
- Almost all MPI calls supported
- Popular on Linux clusters and in the SciPy community
- Operations are primarily methods on communicator objects
- Supports communication of pickled (serializable) Python objects
- Optimized communication of NumPy arrays
- API docs: `https://mpi4py.readthedocs.io/en/stable/`

The mpi4py module

- Python interface to MPI
- Based on MPI-2 C/C++ bindings
- Almost all MPI calls supported
- Popular on Linux clusters and in the SciPy community
- Operations are primarily methods on communicator objects
- Supports communication of pickled (serializable) Python objects
- Optimized communication of NumPy arrays
- API docs: `https://mpi4py.readthedocs.io/en/stable/`

The mpi4py module

- Python interface to MPI
- Based on MPI-2 C/C++ bindings
- Almost all MPI calls supported
- Popular on Linux clusters and in the SciPy community
- Operations are primarily methods on communicator objects
- Supports communication of pickled (serializable) Python objects
- Optimized communication of NumPy arrays
- API docs: `https://mpi4py.readthedocs.io/en/stable/`

The mpi4py module

- Python interface to MPI
- Based on MPI-2 C/C++ bindings
- Almost all MPI calls supported
- Popular on Linux clusters and in the SciPy community
- Operations are primarily methods on communicator objects
- Supports communication of pickled (serializable) Python objects
- Optimized communication of NumPy arrays
- API docs: `https://mpi4py.readthedocs.io/en/stable/`

Installation of mpi4py

Easy to install with Anaconda:

```
$ conda create -n mpi mpi4py numpy scipy
```

or pip

```
$ pip install mpi4py
```

Simple example

helloworlddeprecated.py

```
from mpi4py import MPI
""" Get a communicator:
    The most common communicator is the
    one that connects all available processes
    which is called COMM_WORLD
"""
comm = MPI.COMM_WORLD

# print rank (the process number) and overall number of processes
print("Hello World: process", comm.Get_rank(), " out of
    ",comm.Get_size()," is reporting for duty!")
```

On Linux/Mac OS run

```
mpirun -np 4 python helloworld.py
```

On Windows install MSMPI and then set %PATH% environment variable

```
mpiexec /np 4 python helloworld.py
```

to execute the script using 4 processes.

Simple example

helloworld.py

```
from mpi4py import MPI
""" Get a communicator:
    The most common communicator is the
    one that connects all available processes
    which is called COMM_WORLD
"""
# acts like COMM_WORLD but is a separate instance
comm = MPI.Comm.Clone( MPI.COMM_WORLD )

# print rank (the process number) and overall number of processes
print("Hello World: process", comm.Get_rank(), " out of
    ",comm.Get_size()," is reporting for duty!")
```

On Linux/Mac OS run

```
mpirun -np 4 python helloworld.py
```

On Windows install MSMPI and then set %PATH% environment variable

```
mpiexec /np 4 python helloworld.py
```

to execute the script using 4 processes.

Simple example

helloworld.py

```
from mpi4py import MPI
""" Get a communicator:
    The most common communicator is the
    one that connects all available processes
    which is called COMM_WORLD
"""
# acts like COMM_WORLD but is a separate instance
comm = MPI.Comm.Clone( MPI.COMM_WORLD )

# print rank (the process number) and overall number of processes
print("Hello World: process", comm.Get_rank(), " out of
    ",comm.Get_size()," is reporting for duty!")
```

On Linux/Mac OS run

```
mpirun -np 4 python helloworld.py
```

On Windows install MSMPI and then set %PATH% environment variable

```
mpiexec /np 4 python helloworld.py
```

to execute the script using 4 processes.

Note:

- `MPI_Initialize` is called upon `import` `mpi4py` and
- `MPI_Finalize` is called when the script exits

MPI: most basic send and receive commands

sendrecv.py

```
from mpi4py import MPI
import numpy as np
""" Get a communicator:
    The most common communicator is the
    one that connects all available processes
    which is called COMM_WORLD.
    Clone the communicator to avoid interference
    with other libraries or applications
"""
comm = MPI.Comm.Clone( MPI.COMM_WORLD )

rank = comm.Get_rank()
if rank == 0:
    # send 10 numbers to rank 1 (dest=1)
    data = np.array([range(1,10)])
    print("P[",rank,"] sent data =",data)
    # method 'send' for Python objects (pickle under the hood):
    comm.send(data, dest=1 )

if rank == 1:
    # receive 10 numbers from rank 0 (source=0)
    # method 'recv' for Python objects (pickle under the hood):
    data = comm.recv(source=0 )
    print("P[",rank,"] received data =",data)
```

MPI: optimized Send/Recv for numpy arrays

SendRecv.py

```
from mpi4py import MPI
import numpy as np
""" Get a communicator:
    The most common communicator is the
    one that connects all available processes
    which is called COMM_WORLD.
    Clone the communicator to avoid interference
    with other libraries or applications
"""
comm = MPI.Comm.Clone( MPI.COMM_WORLD )

rank = comm.Get_rank()
if rank == 0:
    # send 10 numbers to rank 1 (dest=1)
    data = np.arange(10, dtype='f')
    comm.Send([data,MPI.INT], dest=1, tag=42 )
if rank == 1:
    # receive 10 numbers from rank 0 (source=0)
    data = np.empty(10, dtype='f')
    comm.Recv(data, source=0, tag=42 )
    print("P[",rank,"] received data =",data)
```

MPI: commonly made mistakes

failure.py

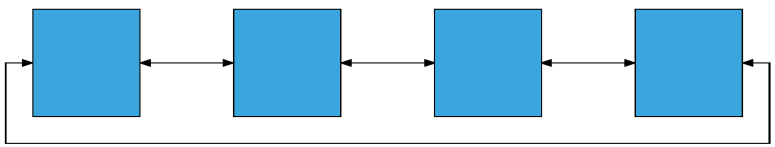
```
from mpi4py import MPI
comm = MPI.Comm.Clone( MPI.COMM_WORLD )
rank = comm.Get_rank()

import numpy as np
data = np.array([range(1,10)])

""" Make sure that every send has a matching recv! """

if rank == 0:
    # send 10 numbers to rank 1 (dest=1)
    # method 'send' for Python objects (pickle under the hood):
    comm.send(data, dest=1 )

if np.linalg.norm( data ) > 1:
    # receive 10 numbers from rank 0 (source=0)
    # method 'recv' for Python objects (pickle under the hood):
    data = comm.recv(source=0 )
    print("P[",rank,"] received data =",data)
```

Exercise: Let's program a ring communication

Exercise: Lets program a ring where every process communicates with the neighboring processes only.
Let r be the rank of each process for $P \in I\!N$ proceses. Compute the sum of all ranks:

$$s = \sum_{i=0}^{P-1} r_i$$

Use the above mentioned `send` and `recv` commands.



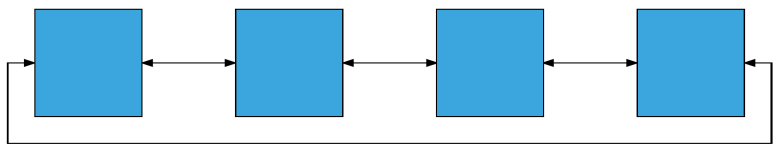
1d torus (from Wikipedia)

Exercise: Let's program a ring communication

Exercise: Lets program a ring where every process communicates with the neighboring processes only.
Let r be the rank of each process for $P \in I\!N$ proceses. Compute the sum of all ranks:

$$s = \sum_{i=0}^{P-1} r_i$$

Use the above mentioned `send` and `recv` commands.



1d torus (from Wikipedia)

Compare your result with the simple call

allreduce.py

```
s = comm.allreduce( rank, op=MPI.SUM )
```

Further Reading

mpi4py
https://mpi4py.readthedocs.io/en/stable/
Accessed 2021.

Open MPI
`https://www.open-mpi.org/`.
Accessed 2021.

MPI Forum
https://www.mpi-forum.org
Accessed 2021.