Implementation of Parallelization

Part II - Multiprocessing: Python Multiprocessing & MPI

Jascha Schewtschenko

Royal Observatory of Edinburgh, University of Edinburgh

May 16, 2025

Outline

Python/multiprocessing

Python/multiprocessing

Python/multiprocessing

- First introduced for Python 2.6

Python/multiprocessing

- First introduced for Python 2.6
- Goals:

  Simplicity The `multiprocessing` module provides a functionality VERY similar to the *Threading* module, but is avoiding the GIL issue

  Portability Like `threading`, it is supported on a variety of implementations/platforms/OS

Python/multiprocessing

- First introduced for Python 2.6
- Goals:

  Simplicity The `multiprocessing` module provides a functionality VERY similar to the *Threading* module, but is avoiding the GIL issue

  Portability Like `threading`, it is supported on a variety of implementations/platforms/OS
- spawns separate processes with their own memory, but also separate GILs

Python/multiprocessing

- First introduced for Python 2.6
- Goals:
  - Simplicity: The `multiprocessing` module provides a functionality VERY similar to the *Threading* module, but is avoiding the GIL issue
  - Portability: Like `threading`, it is supported on a variety of implementations/platforms/OS
- spawns separate processes with their own memory, but also separate GILs
- allows to parallelise CPU-heavy work efficiently

Python/multiprocessing - Process management: Creation & Term.

- Analogue to Python threads, processes are created explicitly by creating an instance of the `Process` class (`multiprocessing.Process(target=...[,args=...][,name=...],...)`)

Python/multiprocessing - Process management: Creation & Term.

- Analogue to Python threads, processes are created explicitly by creating an instance of the `Process` class (`multiprocessing.Process(target=...[,args=...][,name=...],...)`)
- They terminate when finishing their starting routine or, if 'daemonic' once the main program finishes

Python/multiprocessing - Process management: Creation & Term.

- Analogue to Python threads, processes are created explicitly by creating an instance of the `Process` class (`multiprocessing.Process(target=...[,args=...][,name=...],...)`)
- They terminate when finishing their starting routine or, if 'daemonic' once the main program finishes
- You can explicitly wait for a process to finish by invoking its *join()* method

Python/multiprocessing - Process management: Creation & Term.

- Analogue to Python threads, processes are created explicitly by creating an instance of the `Process` class (`multiprocessing.Process(target=...[,args=...][,name=...],...)`)
- They terminate when finishing their starting routine or, if 'daemonic' once the main program finishes
- You can explicitly wait for a process to finish by invoking its *join()* method
- Unlike basic `Thread` objects, `Process` instances can be terminated using their `terminate()` method

Python/multiprocessing - Process management: Creation & Term.

- Analogue to Python threads, processes are created explicitly by creating an instance of the `Process` class (`multiprocessing.Process(target=...[,args=...][,name=...],...)`)
- They terminate when finishing their starting routine or, if 'daemonic' once the main program finishes
- You can explicitly wait for a process to finish by invoking its *join()* method
- Unlike basic `Thread` objects, `Process` instances can be terminated using their `terminate()` method
- The `Pool` class provides a convenient create one process with a specific function for each element of an input array (with barrier at end like *do/for loop* in OpenMP (`map()`) or non-blocking (`map_async()`))

Python/multiprocessing: Example 1 (Thread vs Process)

```
import threading
import time
import random

def do_something():

   time.sleep(random.randrange(1,5))
   print('%s RUNNING\n' % (threading.current_thread().name))

if __name__ == '__main__':

    threads = []

    for i in range(10):
        my_thread = threading.Thread(target=do_something)
        threads.append(my_thread)

    for t in threads:
        t.start()

    for t in threads:
        t.join()

    print('Done')
```

Python/multiprocessing: Example 1 (Thread vs Process)

```python
import threading
import time
import random

def do_something():

   time.sleep(random.randrange(1,5))
   print('%s RUNNING\n' % (threading.current_thread().name))

if __name__ == '__main__':

    threads = []

    for i in range(10):
        my_thread = threading.Thread(target=do_something)
        threads.append(my_thread)

    for t in threads:
        t.start()

    for t in threads:
        t.join()

    print('Done')
```

```python
import multiprocessing
import time
import random

def do_something():

   time.sleep(random.randrange(1,5))
   print('%s RUNNING\n' % (multiprocessing.current_process().name))

if __name__ == '__main__':

    threads = []

    for i in range(10):
        my_proc = multiprocessing.Process(target=do_something)
        procs.append(my_proc)

    for p in procs:
        p.start()

    for p in procs:
        p.join()

    print('Done')
```

Python/multiprocessing: Example 2 (Process/Pool)

```
import multiprocessing
import time
import random

def do_something(value):

   time.sleep(random.randrange(1,5))
   print('%s RUNNING with value %d\n'
      % (multiprocessing.current_process().name, value))

if __name__ == '__main__':

    threads = []

    val = [pow(2,i) for i in range(10)]

    for i in range(10):
        my_proc = multiprocessing.Process(
              target=do_something,args=(val[i],))
        procs.append(my_proc)

    for p in procs:
        p.start()

    for p in procs:
        p.join()

    print('Done')
```

Python/multiprocessing: Example 2 (Process/Pool)

```python
import multiprocessing
import time
import random

def do_something(value):

    time.sleep(random.randrange(1,5))
    print('%s RUNNING with value %d\n'
      % (multiprocessing.current_process().name, value))

if __name__ == '__main__':

    threads = []

    val = [pow(2,i) for i in range(10)]

    for i in range(10):
        my_proc = multiprocessing.Process(
            target=do_something,args=(val[i],))
        procs.append(my_proc)

    for p in procs:
        p.start()

    for p in procs:
        p.join()

    print('Done')
```

```python
import multiprocessing
import time
import random

def do_something(value):

    time.sleep(random.randrange(1,5))
    print('%s RUNNING with value %d\n'
      % (multiprocessing.current_process().name, value))

if __name__ == '__main__':

    values = [pow(2,i) for i in range(10)]

    pool = multiprocessing.Pool(10)
    pool.map(do_something,values)

    pool.close()
    pool.join()

    print('Done')
```

Python/multiprocessing: Synchronization / Flow control

- The same control structures as for `threading` exist for the `multiprocessing` module:

  `Barrier` (Python3 only)
  `Lock/RLock`
  `Condition`
  `Semaphore`

Python/multiprocessing: Example 3 (Shared ctypes)

```
import multiprocessing
import threading

def do_something():

   global shared_value

   print('BEFORE shared_value = {}'.format(shared_value))
   shared_value += 1
   print('AFTER shared value = {}\n'.format(shared_value))

if __name__ == '__main__':

    global shared_value
    shared_value = 0

    print('INITIAL: shared_value={}\n'.format(shared_value))

    my_proc = multiprocessing.Process(target=do_something)
    my_proc.start()
    my_proc.join()

    print('PROCESS: shared_value={}\n'.format(shared_value))

    my_thread = threading.Thread(target=do_something)
    my_thread.start()
    my_thread.join()

    print('THREAD shared_value={}\n'.format(shared_value))

    print('Done')
```

Python/multiprocessing: Example 3 (Shared ctypes)

```
import multiprocessing
import threading

def do_something():

   global shared_value

   print('BEFORE shared_value = {}'.format(shared_value))
   shared_value += 1
   print('AFTER shared value = {}\n'.format(shared_value))

if __name__ == '__main__':

    global shared_value
    shared_value = 0

    print('INITIAL: shared_value={}\n'.format(shared_value))

    my_proc = multiprocessing.Process(target=do_something)
    my_proc.start()
    my_proc.join()

    print('PROCESS: shared_value={}\n'.format(shared_value))

    my_thread = threading.Thread(target=do_something)
    my_thread.start()
    my_thread.join()

    print('THREAD: shared_value={}\n'.format(shared_value))

    print('Done')
```

SHARED_VALUE = 1 !!!

Python/multiprocessing: Memory management

- since processes do **NOT** share the same memory, i.e. global variables are **NOT** shared shared across processes(!)

Python/multiprocessing: Memory management

- since processes do **NOT** share the same memory, i.e. global variables are **NOT** shared shared across processes(!)
- Simple `ctypes` can be stored in special synchronized memory maps (`Value`, `Array`)

Python/multiprocessing: Example 3 (Shared ctypes)

```
import multiprocessing
import threading

def do_something():

   global shared_value

   print('BEFORE shared_value = {}'.format(shared_value))
   shared_value += 1
   print('AFTER shared value = {}\n'.format(shared_value))

if __name__ == '__main__':

    global shared_value
    shared_value = 0

    print('INITIAL: shared_value={}\n'.format(shared_value))

    my_proc = multiprocessing.Process(target=do_something)
    my_proc.start()
    my_proc.join()

    print('PROCESS: shared_value={}\n'.format(shared_value))

    my_thread = threading.Thread(target=do_something)
    my_thread.start()
    my_thread.join()

    print('THREAD: shared_value={}\n'.format(shared_value))

    print('Done')
```

SHARED_VALUE = 1 !!!

Python/multiprocessing: Example 3 (Shared ctypes)

```
import multiprocessing
import threading

def do_something():

   global shared_value

   print('BEFORE shared_value = {}'.format(shared_value))
   shared_value += 1
   print('AFTER shared value = {}\n'.format(shared_value))

if __name__ == '__main__':

    global shared_value
    shared_value = 0

    print('INITIAL: shared_value={}\n'.format(shared_value))

    my_proc = multiprocessing.Process(target=do_something)
    my_proc.start()
    my_proc.join()

    print('PROCESS: shared_value={}\n'.format(shared_value))

    my_thread = threading.Thread(target=do_something)
    my_thread.start()
    my_thread.join()

    print('THREAD shared_value={}\n'.format(shared_value))

    print('Done')
```

SHARED_VALUE = 1 !!!

```
import multiprocessing
import threading

def do_something(shared_value):

   print('BEFORE shared_value = {}'.format(shared_value.value))
   shared_value.value += 1
   print('AFTER shared value = {}\n'.format(shared_value.value))

if __name__ == '__main__':

    shared_value = multiprocessing.Value('i',0)

    print('INITIAL: shared_value={}\n'.format(shared_value.value))

    my_proc = multiprocessing.Process(
                target=do_something,args=(shared_value,))
    my_proc.start()
    my_proc.join()

    print('PROCESS: shared_value={}\n'.format(shared_value.value))

    my_thread = threading.Thread(
                target=do_something,args=(shared_value,))
    my_thread.start()
    my_thread.join()

    print('THREAD shared_value={}\n'.format(shared_value.value))

    print('Done')
```

Python/multiprocessing: Memory management

- since processes do **NOT** share the same memory, i.e. global variables are **NOT** shared across processes(!)
- Simple `ctypes` can be stored in special synchronized memory maps (`Value`, `Array`)

Python/multiprocessing: Memory management

- since processes do **NOT** share the same memory, i.e. global variables are **NOT** shared across processes(!)
- Simple `ctypes` can be stored in special synchronized memory maps (`Value`, `Array`)
- `(Joinable)Queue` objects work identical to `Queue.Queue` for threads (both thread- and process-safe!) and can transfer 'pickleable' Python objects between processes

Python/multiprocessing: Memory management

- since processes do **NOT** share the same memory, i.e. global variables are **NOT** shared across processes(!)
- Simple `ctypes` can be stored in special synchronized memory maps (`Value`, `Array`)
- `(Joinable)Queue` objects work identical to `Queue.Queue` for threads (both thread- and process-safe!) and can transfer 'pickleable' Python objects between processes
- The `Manager` class provides methods to create proxies for `list`, `dict`, `Namespace`, `Lock`, `RLock`, `Semaphore`, `BoundedSemaphore`, `Condition`, `Event`, `Queue`, `Value` and Array; in particular useful for `Pool` where arguments 'pickled'

Python/multiprocessing: Example 4 (JoinableQueue & Manager)

```
import multiprocessing
import time

def do_something(q):

    while True:
        data = q.get()
        time.sleep(1)
        print('data found to be PROCESSED: {}'.format(data))
        q.task_done()


if __name__ == '__main__':

    q = multiprocessing.JoinableQueue()

    data = [pow(2,i) for i in range(10)]

    process_consumer = multiprocessing.Process(
                            target=do_something, args=(q,))
    process_consumer.start()

    for d in data:
        q.put(d)

    q.join()

    process_consumer.terminate()

    print("Done")
```

Python/multiprocessing: Example 4 (JoinableQueue & Manager)

```
import multiprocessing
import time

def do_something(q):

    while True:
        data = q.get()
        time.sleep(1)
        print('data found to be PROCESSED: {}'.format(data))
        q.task_done()


if __name__ == '__main__':

    q = multiprocessing.JoinableQueue()

    data = [pow(2,i) for i in range(10)]

    process_consumer = multiprocessing.Process(
                           target=do_something, args=(q,))
    process_consumer.start()

    for d in data:
        q.put(d)

    q.join()

    process_consumer.terminate()

    print("Done")
```

```
import multiprocessing
import time
import random

def do_something(q):

   while True:
        time.sleep(random.randrange(1,5))
        value = q.get()
        print('%s RUNNING with value %d\n'
           % (multiprocessing.current_process().name, value))
        q.task_done()

if __name__ == '__main__':

    threads = []

    manager = multiprocessing.Manager()

    #queues = [multiprocessing.Queue() for i in range(10)]
    queues = [manager.Queue() for i in range(10)]

    for i in range(10):
        queues[i].put(pow(2,i))

    pool = multiprocessing.Pool(10)
    pool.map_async(do_something,queues)

    for i in range(10):
        queues[i].join()

    print('Done')
```

Python/multiprocessing: Memory management (cont.)

- a very low-level way to communicate/exchange data are with the help `Connection` objects for *Message Passing* in python (which is then used in many higher-level classes like e.g. `Queue`)

Python/multiprocessing: Memory management (cont.)

- a very low-level way to communicate/exchange data are with the help `Connection` objects for *Message Passing* in python (which is then used in many higher-level classes like e.g. `Queue`)
- The `Pipe` function creates a pair of them connected with (duplex) communication pipe between them

Python/multiprocessing: Memory management (cont.)

- a very low-level way to communicate/exchange data are with the help `Connection` objects for *Message Passing* in python (which is then used in many higher-level classes like e.g. `Queue`)
- The `Pipe` function creates a pair of them connected with (duplex) communication pipe between them
- They provide a non-blocking `send(obj)` and a `recv([block])` (default: blocking) method (which are neither thread- nor process-safe). `obj` has to be a 'pickleable' python object

Python/multiprocessing: Memory management (cont.)

- a very low-level way to communicate/exchange data are with the help `Connection` objects for *Message Passing* in python (which is then used in many higher-level classes like e.g. `Queue`)
- The `Pipe` function creates a pair of them connected with (duplex) communication pipe between them
- They provide a non-blocking `send(obj)` and a `recv([block])` (default: blocking) method (which are neither thread- nor process-safe). `obj` has to be a 'pickleable' python object
- Alternatively, there are `send_bytes(buf)` and `recv_bytes()` where `buf` supports the `buffer` interface

Python/multiprocessing: Memory management (cont.)

- a very low-level way to communicate/exchange data are with the help `Connection` objects for *Message Passing* in python (which is then used in many higher-level classes like e.g. `Queue`)
- The `Pipe` function creates a pair of them connected with (duplex) communication pipe between them
- They provide a non-blocking `send(obj)` and a `recv([block])` (default: blocking) method (which are neither thread- nor process-safe). `obj` has to be a 'pickleable' python object
- Alternatively, there are `send_bytes(buf)` and `recv_bytes()` where `buf` supports the `buffer` interface
- The blocking nature of `recv()` may lead to deadlocks(!) (you can use the `poll()` method to check for available data in the pipe)

Python/multiprocessing: Example 5 (Message Passing)

```
import multiprocessing
import time

def do_something(pipe):

   while True:
        value = pipe.recv()
        print('%s RECEIVED: %s\n' %
                        (multiprocessing.current_process().name, value))
        time.sleep(1)
        pipe.send('ACK')

if __name__ == '__main__':

    data = [2,'Test',[1,2,3]]

    pipe1, pipe2 = multiprocessing.Pipe()
    my_proc = multiprocessing.Process(target=do_something,args=(pipe2,))
    my_proc.start()

    for d in data:
        pipe1.send(d)
        value = pipe1.recv()
        print('%s CONFIRMATION received: %s\n' %
                       (multiprocessing.current_process().name, value))

    my_proc.terminate()

    print('Done')
```

MPI

MPI - History/Goals

- Goals:

MPI - History/Goals

- Goals:

  Standardization: De-facto industry “standard” for message passing.

  Portability: Runs on a huge variety of platforms, allows for parallelization on very heterogeneous clusters

MPI - History/Goals

- Goals:

  Standardization De-facto industry “standard” for message passing.

  Portability Runs on a huge variety of platforms, allows for parallelization on very heterogeneous clusters

- First presented at supercomputing conference in 1993, initial releases in 1994 (MPI-1), 1998 (MPI-2), 2012 (MPI-3)

MPI - History/Goals

- Goals:
  - **Standardization** De-facto industry “standard” for message passing.
  - **Portability** Runs on a huge variety of platforms, allows for parallelization on very heterogeneous clusters
- First presented at supercomputing conference in 1993, initial releases in 1994 (MPI-1), 1998 (MPI-2), 2012 (MPI-3)
- Many popular implementations e.g. OpenMPI (free), Intel MPI, MPICH

MPI - History/Goals (Python)

MPI - History/Goals (Python)

- Historically, there are a few python modules that provide an MPI-based multiprocessing framework

MPI - History/Goals (Python)

- Historically, there are a few python modules that provide an MPI-based multiprocessing framework

  PyPar implements scalable parallelism using essential subset of the MPI standard; light-weight & efficient, but lacking some features

  pyMPI/MPI Python [non-active] provides (built-in) MPI module (for cpython) for basic parallel programming

  ScientificPython [≠ SciPy!, non-active] collection of Python modules useful for scientific computing; contains simple, incomplete MPI interface

MPI - History/Goals (Python)

- Historically, there are a few python modules that provide an MPI-based multiprocessing framework

  PyPar
  : implements scalable parallelism using essential subset of the MPI standard; light-weight & efficient, but lacking some features

  pyMPI/MPI Python
  : [non-active] provides (built-in) MPI module (for cpython) for basic parallel programming

  ScientificPython
  : [$\neq$ SciPy!, non-active] collection of Python modules useful for scientific computing; contains simple, incomplete MPI interface

- Nowadays, `mpi4py` is the most popular MPI-based parallelisation framework for python:

MPI - History/Goals (Python)

- Historically, there are a few python modules that provide an MPI-based multiprocessing framework

  PyPar implements scalable parallelism using essential subset of the MPI standard; light-weight & efficient, but lacking some features

  pyMPI/MPI Python [non-active] provides (built-in) MPI module (for cpython) for basic parallel programming

  ScientificPython [$\neq$ SciPy!, non-active] collection of Python modules useful for scientific computing; contains simple, incomplete MPI interface

- Nowadays, `mpi4py` is the most popular MPI-based parallelisation framework for python:
  - Interface VERY similar to MPI-2 standard C bindings $\Rightarrow$ if you know MPI, you understand mpi4py easily

MPI - History/Goals (Python)

- Historically, there are a few python modules that provide an MPI-based multiprocessing framework

  PyPar implements scalable parallelism using essential subset of the MPI standard; light-weight & efficient, but lacking some features

  pyMPI/MPI Python [non-active] provides (built-in) MPI module (for cpython) for basic parallel programming

  ScientificPython [$\neq$ SciPy!, non-active] collection of Python modules useful for scientific computing; contains simple, incomplete MPI interface

- Nowadays, `mpi4py` is the most popular MPI-based parallelisation framework for python:
  - Interface VERY similar to MPI-2 standard C bindings $\Rightarrow$ if you know MPI, you understand mpi4py easily
  - provides some additional, convenient extensions beyond the pure MPI standard

MPI - History/Goals (Python)

- Historically, there are a few python modules that provide an MPI-based multiprocessing framework

  PyPar implements scalable parallelism using essential subset of the MPI standard; light-weight & efficient, but lacking some features

  pyMPI/MPI Python [non-active] provides (built-in) MPI module (for cpython) for basic parallel programming

  ScientificPython [$\neq$ SciPy!, non-active] collection of Python modules useful for scientific computing; contains simple, incomplete MPI interface

- Nowadays, `mpi4py` is the most popular MPI-based parallelisation framework for python:
  - Interface VERY similar to MPI-2 standard C bindings $\Rightarrow$ if you know MPI, you understand mpi4py easily
  - provides some additional, convenient extensions beyond the pure MPI standard
  - under active development (i.e. makes use of newest features in MPI-3 standard)

MPI - Compiling & Running

MPI - Compiling & Running

- For compiling MPI programs, each implementation comes with specific "wrapper" scripts for the compilers, e.g.

		GNU	Intel
OpenMPI	**C**	`mpicc`	
	C++	`mpiCC/mpic++/mpicxx`	
	Fortran	`mpifort`	
Intel MPI	**C**	`mpicc/mpigcc`	`mpicc/mpiicc`
	C++	`mpi{CC,c++,cxx}/mpigxx`	`mpi{CC,c++,cxx}/mpiicpc`
	Fortran	`mpifort`	`mpifort*/mpiifort`

MPI - Compiling & Running

- For compiling MPI programs, each implementation comes with specific "wrapper" scripts for the compilers, e.g.

		GNU	Intel
OpenMPI	**C**	mpicc	
	C++	mpiCC/mpic++/mpicxx	
	Fortran	mpifort	
Intel MPI	**C**	mpicc/mpigcc	mpicc/mpiicc
	C++	mpi{CC,c++,cxx}/mpigxx	mpi{CC,c++,cxx}/mpiicpc
	Fortran	mpifort	mpifort*/mpiifort

- For running a MPI program, we use `mpirun`/`srun`, which starts as many copies of the program as requested on nodes provided by the batch system, e.g.

```
mpirun -np 4 my_program
srun --mpi=pmi2 -n 4 my_program
```
(SLURM)

MPI - Compiling & Running

- For compiling MPI programs, each implementation comes with specific "wrapper" scripts for the compilers, e.g.

		GNU	Intel
OpenMPI	**C**	mpicc	
	C++	mpiCC/mpic++/mpicxx	
	Fortran	mpifort	
Intel MPI	**C**	mpicc/mpigcc	mpicc/mpiicc
	C++	mpi{CC,c++,cxx}/mpigxx	mpi{CC,c++,cxx}/mpiicpc
	Fortran	mpifort	mpifort*/mpiifort

- For running a MPI program, we use `mpirun`/`srun`, which starts as many copies of the program as requested on nodes provided by the batch system, e.g.

```
mpirun -np 4 my_program
srun --mpi=pmi2 -n 4 my_program
```
(SLURM)

For Python:

```
mpirun -np 4 python my_program.py
srun --mpi=pmi2 -n 4 python my_program.py
```
(SLURM)

Hint: if interpreter specified in source file and file is executable (cf. example files), skip python call e.g.

```
mpirun -np 4 my_program.py
```

MPI - Init & Finalize

MPI - Init & Finalize

- Before using any MPI routine (or better as early as possible), the MPI framework must be initialized and, at the end of the program, the MPI execution environment always properly terminated/cleaned up.

MPI - Init & Finalize

- Before using any MPI routine (or better as early as possible), the MPI framework must be initialized and, at the end of the program, the MPI execution environment always properly terminated/cleaned up.
- In C/Fortran:
  - initialisation has to be done explicitly by calling `MPI_Init(&argc,&argv)`, which also broadcast the command line arguments to all processes

MPI - Init & Finalize

- Before using any MPI routine (or better as early as possible), the MPI framework must be initialized and, at the end of the program, the MPI execution environment always properly terminated/cleaned up.
- In C/Fortran:
  - initialisation has to be done explicitly by calling `MPI_Init(&argc,&argv)`, which also broadcast the command line arguments to all processes
  - At the end of your C/Fortran program, always call `MPI_Finalize()`

MPI - Init & Finalize

- Before using any MPI routine (or better as early as possible), the MPI framework must be initialized and, at the end of the program, the MPI execution environment always properly terminated/cleaned up.
- In C/Fortran:
  - initialisation has to be done explicitly by calling `MPI_Init(&argc,&argv)`, which also broadcast the command line arguments to all processes
  - At the end of your C/Fortran program, always call `MPI_Finalize()`
- In Python:
  - `MPI.Init()` is automatically called when loading the `mpi4py` module

MPI - Init & Finalize

- Before using any MPI routine (or better as early as possible), the MPI framework must be initialized and, at the end of the program, the MPI execution environment always properly terminated/cleaned up.
- In C/Fortran:
  - initialisation has to be done explicitly by calling `MPI_Init(&argc,&argv)`, which also broadcast the command line arguments to all processes
  - At the end of your C/Fortran program, always call `MPI_Finalize()`
- In Python:
  - `MPI.Init()` is automatically called when loading the `mpi4py` module (caveat: currently some issues with `mpi4py` on certain intel network infrastructure require to disable unsupported features at import)

```
import mpi4py; mpi4py.rc.recv_mprobe = False
```

MPI - Init & Finalize

- Before using any MPI routine (or better as early as possible), the MPI framework must be initialized and, at the end of the program, the MPI execution environment always properly terminated/cleaned up.
- In C/Fortran:
  - initialisation has to be done explicitly by calling `MPI_Init(&argc,&argv)`, which also broadcast the command line arguments to all processes
  - At the end of your C/Fortran program, always call `MPI_Finalize()`
- In Python:
  - `MPI.Init()` is automatically called when loading the `mpi4py` module (caveat: currently some issues with `mpi4py` on certain intel network infrastructure require to disable unsupported features at import)

    ```
    import mpi4py; mpi4py.rc.recv_mprobe = False
    ```

  - mpi4py also registers `MPI.Finalize()` for automatic execution on exit $\Rightarrow$ no need to be called explicitly.

MPI - Communicators

MPI - Communicators

- MPI uses *communicators* to define which processes may communicate with each other - in many cases, the predefined `MPI_COMM_WORLD` / `MPI.COMM_WORLD`, which includes all MPI processes.

MPI - Communicators

- MPI uses *communicators* to define which processes may communicate with each other - in many cases, the predefined `MPI_COMM_WORLD` / `MPI.COMM_WORLD`, which includes all MPI processes.
- each process has a unique *rank* within the communicator. You can get the rank for a process with the command `MPI_Comm_rank(comm,&rank)` / `rank=comm.Get_rank()` as well as the total size of the communicator (`MPI_Comm_size(comm,&size)` / `size=comm.Get_size()`)

MPI - Example

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

int main(int argc, char *argv[]) {
int  numtasks, rank, len, rc;
char hostname[MPI_MAX_PROCESSOR_NAME];

// initialize MPI
MPI_Init(&argc,&argv);

// get number of tasks
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);

// get my rank
MPI_Comm_rank(MPI_COMM_WORLD,&rank);

// this one is obvious
MPI_Get_processor_name(hostname, &len);
printf ("Number of tasks= %d My rank= %d
 Running on %s\n", numtasks,rank,hostname);

    // do some work with message passing


// done with MPI
MPI_Finalize();
}
```

```python
# load mpi module
import mpi4py; mpi4py.rc.recv_mprobe = False
from mpi4py import MPI

# assign communicator to variable (for convenience)
comm = MPI.COMM_WORLD

# get number of tasks
size = comm.Get_size()

# get my rank
rank = MPI.COMM_WORLD.Get_rank()

# this one is obvious
hostname = MPI.Get_processor_name()
print("Number of tasks=",size," My rank=",rank,
      " Running on",hostname)

    # do some more work with message passing
```

MPI - Communication

MPI - Communication

- In MPI there are routines for Point-to-Point communication (i.e. from one process to exactly one other) as well as for collective communication

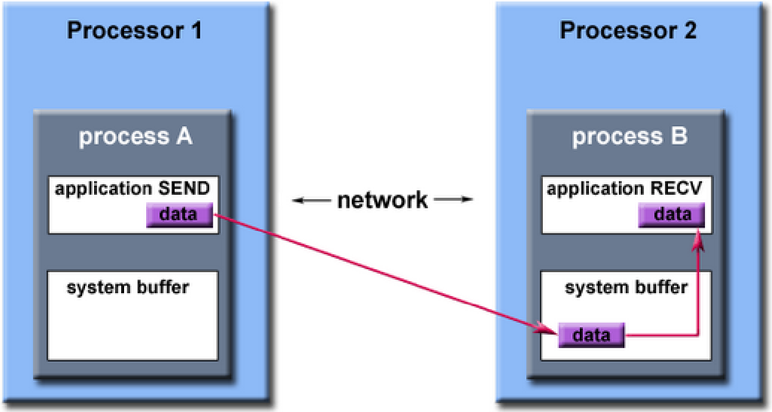
MPI - Communication

- In MPI there are routines for Point-to-Point communication (i.e. from one process to exactly one other) as well as for collective communication
- a Point-to-Point communication always consists of a *send* and a matching *receive* (or combined send/recv) routines

MPI - Communication

- In MPI there are routines for Point-to-Point communication (i.e. from one process to exactly one other) as well as for collective communication
- a Point-to-Point communication always consists of a *send* and a matching *receive* (or combined send/recv) routines
- those routines can be *blocking* and *non-blocking*, *non-synchronous* and *synchronous*

MPI - Communication: Buffering



Path of a message buffered at the receiving process

MPI - Communication: Blocking vs Non-Blocking

Blocking A blocking send (`MPI_Send` / `comm.Send`) waits until message is processed by local MPI (does not mean, that message has been received by other processes!), for waiting for confirmed processing by recipient, use synchronous blocking send (`MPI_Ssend` / `comm.Ssend`); a blocking receive waits until data is received and ready for use

Non-Blocking Non-blocking send/receive routines (`MPI_Isend` / `comm.Isend`, `MPI_Irecv` / `comm.Irecv`, `MPI_Issend` / `comm.Issend`) work like their blocking counter-parts, but only request the operation and do not wait for its completion. Instead they return a *request* object that can be used to test/wait (e.g. `MPI_Wait` / `MPI.Request.Wait`, `MPI_Probe` / `comm.Probe`) until operation has been processed/certain status is reached for one or more request simultaneously.

MPI - Communication: Syntax

```
MPI_Isend(&buf,count,datatype,dest,tag,comm,&request)
request=comm.Isend([buf,count*,datatype],dest=<dest>,tag=<tag>)
MPI_Recv(&buf,count,datatype,src,tag,comm,&status)
comm.Recv([buf,count*,datatype],source=<src>,tag=<tag>,status=<statu
```

MPI - Communication: Syntax

```
MPI_Isend(&buf,count,datatype,dest,tag,comm,&request)
request=comm.Isend([buf,count*,datatype],dest=<dest>,tag=<tag>)
MPI_Recv(&buf,count,datatype,src,tag,comm,&status)
comm.Recv([buf,count*,datatype],source=<src>,tag=<tag>,status=<statu
```

with

buf Memory block (e.g. array) to send/receive data from/to

MPI - Communication: Syntax

```
MPI_Isend(&buf,count,datatype,dest,tag,comm,&request)
request=comm.Isend([buf,count*,datatype],dest=<dest>,tag=<tag>)
MPI_Recv(&buf,count,datatype,src,tag,comm,&status)
comm.Recv([buf,count*,datatype],source=<src>,tag=<tag>,status=<statu
```

with

- **buf** Memory block (e.g. array) to send/receive data from/to
- **count** Number of data elements to be sent / max.number to be recv (cf. `MPI_Get_count(..)`/`status.Get_elements(..)`)

MPI - Communication: Syntax

```
MPI_Isend(&buf,count,datatype,dest,tag,comm,&request)
request=comm.Isend([buf,count*,datatype],dest=<dest>,tag=<tag>)
MPI_Recv(&buf,count,datatype,src,tag,comm,&status)
comm.Recv([buf,count*,datatype],source=<src>,tag=<tag>,status=<statu
```

with

- **buf** Memory block (e.g. array) to send/receive data from/to
- **count** Number of data elements to be sent / max.number to be recv (cf. `MPI_Get_count(..)`/`status.Get_elements(..)`)
- **datatype** One of the predefined elementary/derived MPI data types

MPI - Communication: Syntax

C Data Types	
`MPI_CHAR`	char
`MPI_WCHAR`	wchar_t - wide character
`MPI_SHORT`	signed short int
`MPI_INT`	signed int
`MPI_LONG`	signed long int
`MPI_LONG_LONG_INT` `MPI_LONG_LONG`	signed long long int
`MPI_SIGNED_CHAR`	signed char
`MPI_UNSIGNED_CHAR`	unsigned char
`MPI_UNSIGNED_SHORT`	unsigned short int
`MPI_UNSIGNED`	unsigned int
`MPI_UNSIGNED_LONG`	unsigned long int
`MPI_UNSIGNED_LONG_LONG`	unsigned long long int
`MPI_FLOAT`	float
`MPI_DOUBLE`	double
`MPI_LONG_DOUBLE`	long double
`MPI_C_COMPLEX` `MPI_C_FLOAT_COMPLEX`	float _Complex
`MPI_C_DOUBLE_COMPLEX`	double _Complex
`MPI_C_LONG_DOUBLE_COMPLEX`	long double _Complex
`MPI_C_BOOL`	_Bool

`MPI_INT8_T`
`MPI_INT16_T`
`MPI_INT32_T`
`MPI_INT64_T`
	int8_t
	int16_t
	int32_t
	int64_t

`MPI_UINT8_T`
`MPI_UINT16_T`
`MPI_UINT32_T`
`MPI_UINT64_T`
	uint8_t
	uint16_t
	uint32_t
	uint64_t

`MPI_BYTE`	8 binary digits
`MPI_PACKED`	data packed or unpacked with MPI_Pack()/ MPI_Unpack

MPI - Communication: Syntax

```
MPI_Isend(&buf,count,datatype,dest,tag,comm,&request)
request=comm.Isend([buf,count*,datatype],dest=<dest>,tag=<tag>)
MPI_Recv(&buf,count,datatype,src,tag,comm,&status)
comm.Recv([buf,count*,datatype],source=<src>,tag=<tag>,status=<statu
```

with

- **buf** Memory block (e.g. array) to send/receive data from/to
- **count** Number of data elements to be sent / max.number to be recv (cf. `MPI_Get_count(..)`/`status.Get_elements(..)`)
- **datatype** One of the predefined elementary/derived MPI data types
- **dest/src** Rank of the communication partner (within the used shared communicator); wildcard `MPI_ANY_SOURCE`/`MPI.ANY_SOURCE`

MPI - Communication: Syntax

```
MPI_Isend(&buf,count,datatype,dest,tag,comm,&request)
request=comm.Isend([buf,count*,datatype],dest=<dest>,tag=<tag>)
MPI_Recv(&buf,count,datatype,src,tag,comm,&status)
comm.Recv([buf,count*,datatype],source=<src>,tag=<tag>,status=<statu
```

with

- **buf** Memory block (e.g. array) to send/receive data from/to
- **count** Number of data elements to be sent / max.number to be recv (cf. `MPI_Get_count(..)`/`status.Get_elements(..)`)
- **datatype** One of the predefined elementary/derived MPI data types
- **dest/src** Rank of the communication partner (within the used shared communicator); wildcard `MPI_ANY_SOURCE`/`MPI.ANY_SOURCE`
- **tag** arbitrary non-negative (short) integer; same for send & recv (unless wildcard `MPI_ANY_TAG`/`MPI.ANY_TAG` used for recv)

MPI - Communication: Syntax

```
MPI_Isend(&buf,count,datatype,dest,tag,comm,&request)
request=comm.Isend([buf,count*,datatype],dest=<dest>,tag=<tag>)
MPI_Recv(&buf,count,datatype,src,tag,comm,&status)
comm.Recv([buf,count*,datatype],source=<src>,tag=<tag>,status=<statu
```

with

buf	Memory block (e.g. array) to send/receive data from/to
count	Number of data elements to be sent / max.number to be recv (cf. `MPI_Get_count(..)`/`status.Get_elements(..)`)
datatype	One of the predefined elementary/derived MPI data types
dest/src	Rank of the communication partner (within the used shared communicator); wildcard `MPI_ANY_SOURCE`/`MPI.ANY_SOURCE`
tag	arbitrary non-negative (short) integer; same for send & recv (unless wildcard `MPI_ANY_TAG`/`MPI.ANY_TAG` used for recv)
comm	communicator

MPI - Communication: Syntax

```
MPI_Isend(&buf,count,datatype,dest,tag,comm,&request)
request=comm.Isend([buf,count*,datatype],dest=<dest>,tag=<tag>)
MPI_Recv(&buf,count,datatype,src,tag,comm,&status)
comm.Recv([buf,count*,datatype],source=<src>,tag=<tag>,status=<statu
```

with

buf	Memory block (e.g. array) to send/receive data from/to
count	Number of data elements to be sent / max.number to be recv (cf. `MPI_Get_count(..)`/`status.Get_elements(..)`)
datatype	One of the predefined elementary/derived MPI data types
dest/src	Rank of the communication partner (within the used shared communicator); wildcard `MPI_ANY_SOURCE`/`MPI.ANY_SOURCE`
tag	arbitrary non-negative (short) integer; same for send & recv (unless wildcard `MPI_ANY_TAG/MPI.ANY_TAG` used for recv)
comm	communicator
request	allocated request structure used to communicate progress of comm. process for non-blocking routines

MPI - Communication: Syntax

```
MPI_Isend(&buf,count,datatype,dest,tag,comm,&request)
request=comm.Isend([buf,count*,datatype],dest=<dest>,tag=<tag>)
MPI_Recv(&buf,count,datatype,src,tag,comm,&status)
comm.Recv([buf,count*,datatype],source=<src>,tag=<tag>,status=<statu
```

with

- **buf** Memory block (e.g. array) to send/receive data from/to
- **count** Number of data elements to be sent / max.number to be recv (cf. `MPI_Get_count(..)`/`status.Get_elements(..)`)
- **datatype** One of the predefined elementary/derived MPI data types
- **dest/src** Rank of the communication partner (within the used shared communicator); wildcard `MPI_ANY_SOURCE`/`MPI.ANY_SOURCE`
- **tag** arbitrary non-negative (short) integer; same for send & recv (unless wildcard `MPI_ANY_TAG`/`MPI.ANY_TAG` used for recv)
- **comm** communicator
- **request** allocated request structure used to communicate progress of comm. process for non-blocking routines
- **status** allocated status structure containing source & tag for receive

MPI - Communication (Python)

- For convenience (and to avoid deadlocks, cf. exercises), Python also provides a combined send-receive function

```
comm.Sendrecv(send_buf,recv_buf=<recvbuf>,
        source=<src>,dest=<dest>)
```

MPI - Communication (Python)

- For convenience (and to avoid deadlocks, cf. exercises), Python also provides a combined send-receive function

```
comm.Sendrecv(send_buf,recv_buf=<recvbuf>,
    source=<src>,dest=<dest>)
```

- Additionally, Python's `mpi4py` also provides functions to send/receive generic/Pickle-able Python Data Objects instead of buffer-like objects (function names same except for a lower case first letter) e.g.

```
comm.isend(obj,dest=<dest>,tag=<tag>
obj=comm.recv(source=<src>,status=<status>)
```

MPI - Communication (Python)

- For convenience (and to avoid deadlocks, cf. exercises), Python also provides a combined send-receive function

```
comm.Sendrecv(send_buf,recv_buf=<recvbuf>,
              source=<src>,dest=<dest>)
```

- Additionally, Python's `mpi4py` also provides functions to send/receive generic/Pickle-able Python Data Objects instead of buffer-like objects (function names same except for a lower case first letter) e.g.

```
comm.isend(obj,dest=<dest>,tag=<tag>
obj=comm.recv(source=<src>,status=<status>)
```

- `mpi4py` also supports dynamic process management (cf. `MPI.Intracomm.Spawn(..)` & `comm.Disconnect()`)

MPI - Communication: Example

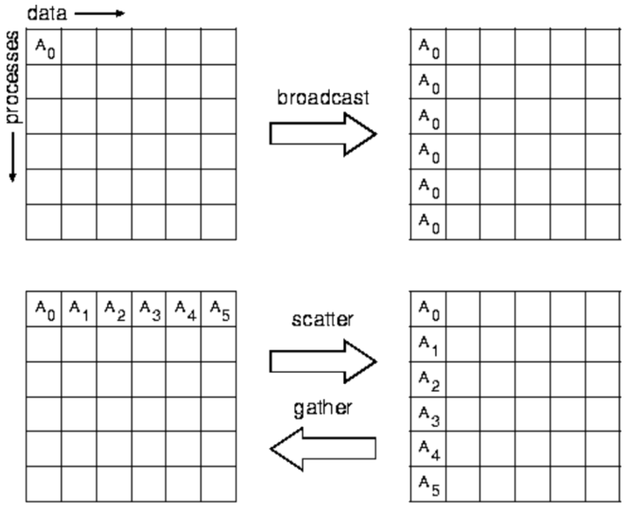
```c
#include <mpi.h>
#include <stdio.h>
#include <math.h>

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

  int numtasks, rank, secret=0, root=0;
  int tag, dest, src;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank == root) {
    secret = 42;
    // send secret to all process, but one
    MPI_Request req;
    for (dest=0;dest<numtasks-1;dest++) {
      tag=dest;
      MPI_Isend(&secret,1,MPI_INT,dest,tag,
            MPI_COMM_WORLD,&req);
    }
  } else if (rank != numtasks-1) {
    // receive secret
    MPI_Status info;
    MPI_Recv(&secret,1,MPI_INT,
      MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&info);
    printf("process %d received data: from %d \
      with tag %d\n",rank,info.MPI_SOURCE,info.MPI_TAG);
  }

  // wait for all processes
  MPI_Barrier(MPI_COMM_WORLD);

  printf("rank %d secret: %d\n",rank,secret);

  MPI_Finalize();
}
```

```python
import numpy as np
import mpi4py; mpi4py.rc.recv_mprobe = False
from mpi4py import MPI

root = 0

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

secret = np.zeros(1, dtype='i')

if (rank == root):
  secret[0] = 42
  # send secret to all process, but one
  for dest in range(1,numtasks-1):
    req=comm.Isend([secret,1,MPI.INT],dest=dest,tag=dest)

elif (rank != numtasks-1):
  info = MPI.Status()
  # receive secret
  comm.Recv(secret,source=MPI.ANY_SOURCE,
           tag=MPI.ANY_TAG,status=info)
  print("process",rank,"received data:",
        "from",info.Get_source(),"with tag",info.Get_tag())

# wait for all processes
comm.Barrier()

print("process",rank,"secret:",secret)
```

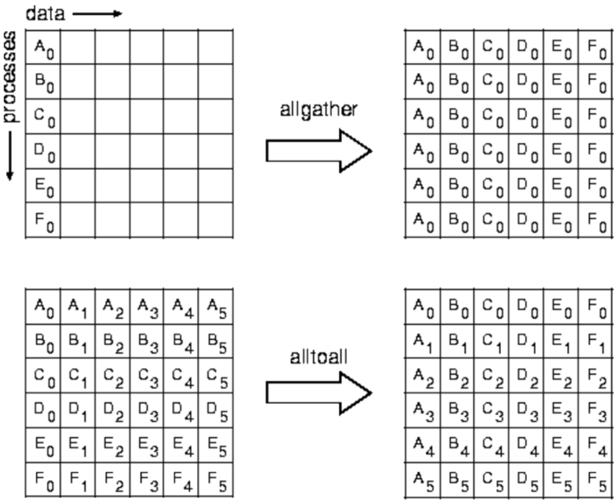
MPI - Collective Communication

- More efficient data exchange with multiple processes
- always involves all processes in one communicator
- can only used with predefined datatypes
- can be blocking or non-blocking (since MPI-3)

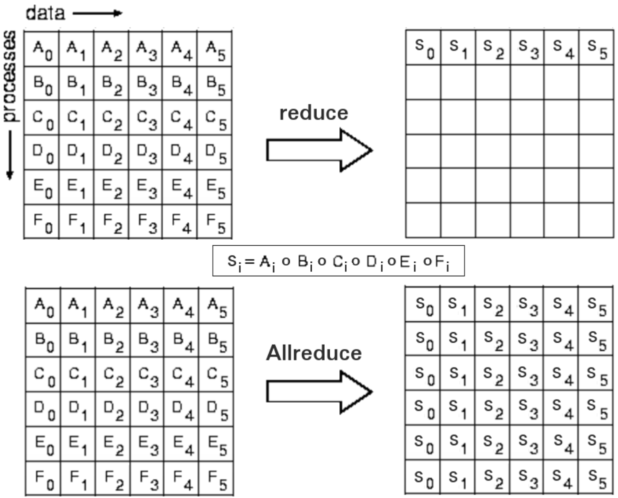
MPI - Collective Comm. [Broadcast/Scatter/Gather]



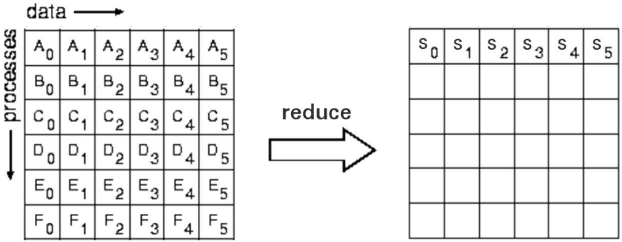
MPI - Collective Communication [Allgather,Alltoall]



MPI - Collective Communication [Reduce,Allreduce]



MPI - Collective Communication [Reduce,Allreduce]



$S_i = A_i \circ B_i \circ C_i \circ D_i \circ E_i \circ F_i$

MPI Reduction Operation		C Data Types
MPI_MAX	maximum	integer, float
MPI_MIN	minimum	integer, float
MPI_SUM	sum	integer, float
MPI_PROD	product	integer, float
MPI_LAND	logical AND	integer
MPI_BAND	bit-wise AND	integer, MPI_BYTE
MPI_LOR	logical OR	integer
MPI_BOR	bit-wise OR	integer, MPI_BYTE
MPI_LXOR	logical XOR	integer
MPI_BXOR	bit-wise XOR	integer, MPI_BYTE
MPI_MAXLOC	max value and location	float, double and long double
MPI_MINLOC	min value and location	float, double and long double

MPI - Collective Communication: Example

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

#define N 100
#define MIN(X, Y) (((X) < (Y)) ? (X) : (Y))

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

  int numtasks, rank, n=0, i, root=0, chunk;
  int a[N], b[N], result=0, final_result;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank == root)
     for (i=0; i<N; i++) {a[i] = i;}
  // broadcast array to all processes
  MPI_Bcast(a,N,MPI_INT,root,MPI_COMM_WORLD);

  // "suboptimal" partitioning
  chunk = ceil((float) N / numtasks);

  for (i=rank*chunk;i<MIN(N,(rank+1)*chunk);i++) {
    result += a[i] * a[i]; n++;
  }

  printf("process %d chunk size: %d\n",rank,n);
  // collect results & sum them up
  MPI_Reduce(&result,&final_result,1,MPI_INT,
             MPI_SUM,root,MPI_COMM_WORLD);

  if (rank ==  root) {
    printf("result= %d\n",final_result);
  }

  MPI_Finalize();
}
```

```python
import math
import numpy as np
import mpi4py; mpi4py.rc.recv_mprobe = False
from mpi4py import MPI

N = 100; n = 0; root = 0
result = np.zeros(1, dtype='i')
final_result = np.zeros(1, dtype='i')

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

if (rank == root):
  a = np.arange(N, dtype='i')
else:
  a = np.zeros(N, dtype='i')

# broadcast array to all processes
comm.Bcast([a,MPI.INT],root=root)

# "suboptimal" partitioning
chunk = math.ceil(float(N) / numtasks);

for i in range(rank*chunk,min(N,(rank+1)*chunk)):
    result[0] += a[i] * a[i]; n+=1

print("process",rank,"chunk size:",n)

# collect results & sum them up
comm.Reduce(result,final_result,
            op=MPI.SUM,root=root)

if (rank ==  root):
  print("result=",final_result)
```

MPI - Multithreading

- As shown in the 'Introduction' talk, you can combine Multithreading and Multiprocessing, **BUT** ...
- you have to check whether your MPI implementations is thread-safe. MPI libraries vary in their level of thread support:

  `MPI_THREAD_SINGLE` no multithreading supported

  `MPI_THREAD_FUNNELED` only main thread may make MPI calls

  `MPI_THREAD_SERIALIZED` MPI calls are serialized i.e. cannot be processed concurrently

  `MPI_THREAD_MULTIPLE` thread-safe

Debugging

Debugging

Debugging

- As for analyzing and tuning parallel program performance, debugging can be much more challenging for parallel programs than for serial programs (in particular for MPI programs)

Debugging

- As for analyzing and tuning parallel program performance, debugging can be much more challenging for parallel programs than for serial programs (in particular for MPI programs)
- And again, unfortunately, covering this topic in any detail would go beyond the scope of this introduction to parallel program.

Debugging

- As for analyzing and tuning parallel program performance, debugging can be much more challenging for parallel programs than for serial programs (in particular for MPI programs)
- And again, unfortunately, covering this topic in any detail would go beyond the scope of this introduction to parallel program.
- While popular open source debuggers like `gdb` provide facilites for debugging multi-threaded programs, MPI debugging relies on commercial solutions like DDT or TotalView