Outline

1. Python tools for parallel computing

2. Parallel Python
   - What is PP?
   - API

3. MPI for Python
   - MPI
   - mpi4py

4. GPU computing with Python
   - GPU computing
   - CUDA
   - PyCUDA
   - Anaconda Accelerate - Numbapro
Symmetric multiprocessing

- Multiprocessing: included in the standard library.
- Parallel Python.
- IPython.
- Others: POSH, pprocess, etc...
Cluster computing

- Message Passing Interface (MPI): mpi4py, pyMPI, pypar, ...
- Parallel Virtual Machine (PVM): pypvm, pynpvm, ...
- IPython.
- Others: Pyro, ScientificPython, ...
Parallel GPU computing

- PyCUDA.
- PyOpenCL.
- Copperhead.
- Anaconda Accelerate.
1. Python tools for parallel computing

2. Parallel Python
   What is PP?
   API

3. MPI for Python
   MPI
   mpi4py

4. GPU computing with Python
   GPU computing
   CUDA
   PyCUDA
   Anaconda Accelerate - Numbapro

Álvaro Leitao Rodríguez (TU Delft)
Parallel Python - PP

- PP is a python module.
- Parallel execution of python code on SMP and clusters.
- Easy to convert serial application in parallel.
- Automatic detection of the optimal configuration.
- Dynamic processors allocation (number of processes can be changed at runtime).
- Cross-platform portability and interoperability (Windows, Linux, Unix, Mac OS X).
- Cross-architecture portability and interoperability (x86, x86-64, etc.).
- Open source: http://www.parallelpython.com/.
Python tools for parallel computing

Parallel Python
  What is PP?
  API

MPI for Python
  MPI
  mpi4py

GPU computing with Python
  GPU computing
  CUDA
  PyCUDA
  Anaconda Accelerate - Numbapro
PP - module API

- Idea: Server provide you workers (processors).
- Workers do a job.
- class **Server** - Parallel Python SMP execution server class
  
  - **init**(self, ncpus='autodetect', ppservers=(), secret=None, restart=False, proto=2, socket_timeout=3600)
  - submit(self, func, args=(), depfuncs=(), modules=(), callback=None, callbackargs=(), group='default', globals=None)
  - Other: get_ncpus, set_ncpus, print_stats, ...

- class **Template**
  
  - **init**(self, job_server, func, depfuncs=(), modules=(), callback=None, callbackargs=(), group='default', globals=None)
  - submit(self, *args)
PP - Examples

• First example: `pp_hello_world.py`
• More useful example: `pp_sum_primes_ntimes.py`
  • What happens if $n$ is too different?
• A really useful example: `pp_sum_primes.py`
  • How long is the execution with different amount of workers?
• Template example: `pp_sum_primes_ntimes_Template.py`
• More involved examples: `pp_montecarlo_pi.py` and `pp_midpoint_integration.py`
1. Python tools for parallel computing
2. Parallel Python
   What is PP?
   API
3. MPI for Python
   MPI
   mpi4py
4. GPU computing with Python
   GPU computing
   CUDA
   PyCUDA
   Anaconda Accelerate - Numbapro
What is MPI?

- An interface specification: MPI = Message Passing Interface.
- MPI is a specification for the developers and users of message passing libraries.
- But, by itself, it is NOT a library (it is the specification of what such a library should be).
- MPI primarily follows the message-passing parallel programming model.
- The interface attempts to be: practical, portable, efficient and flexible.
- Provide virtual topology, synchronization, and communication functionality between a set of processes.
- Today, MPI implementations run on many hardware platforms: Distributed memory, Shared memory, Hybrid, ...
MPI concepts

- MPI processes.
- Communicator: connect groups of processes.
- Communication:
  - Point-to-point:
    - Synchronous: MPI_Send, MPI_Recv.
    - Asynchronous: MPI_Isend, MPI_Recv.
  - Collective: MPI_Bcast, MPI_Reduce, MPI_Gather, MPI_Scatter.
- Rank: within a communicator, every process has its own unique, integer identifier.
1. Python tools for parallel computing
2. Parallel Python
   What is PP?
   API
3. MPI for Python
   MPI
   mpi4py
4. GPU computing with Python
   GPU computing
   CUDA
   PyCUDA
   Anaconda Accelerate - Numbapro
mpi4py

- Python implementation of MPI.
- API based on the standard MPI-2 C++ bindings.
- Almost all MPI calls are supported.
- Code is easy to write, maintain and extend.
- Faster than other solutions (mixed Python and C codes).
- A *pythonic* API that runs at C speed.
- Open source: http://mpi4py.scipy.org/
mpi4py - Basic functions

• Python objects.
  • `send`(self, obj, int dest=0, int tag=0)
  • `recv`(self, obj, int source=0, int tag=0, Status status=None)
  • `bcast`(self, obj, int root=0)
  • `reduce`(self, sendobj, recvobj, op=SUM, int root=0)
  • `scatter`(self, sendobj, recvobj, int root=0)
  • `gather`(self, sendobj, recvobj, int root=0)

• C-like structures.
  • `Send`(self, buf, int dest=0, int tag=0)
  • `Recv`(self, buf, int source=0, int tag=0, Status status=None)
  • `Bcast`(self, buf, int root=0)
  • `Reduce`(self, sendbuf, recvbuf, Op op=SUM, int root=0)
  • `Scatter`(self, sendbuf, recvbuf, int root=0)
  • `Gather`(self, sendbuf, recvbuf, int root=0)
mpi4py - Examples

- First example: mpi_hello_world.py
- Message passing example: mpi_simple.py
- Point-to-point example: mpi_buddy.py
- Collective example: mpi_matrix_mul.py
- Reduce example: mpi_midpoint_integration.py
Python tools for parallel computing

Parallel Python
What is PP?
API

MPI for Python
MPI
mpi4py

GPU computing with Python
GPU computing
CUDA
PyCUDA
Anaconda Accelerate - Numbapro
What is GPU computing?

- GPU computing is the use of a graphics processing unit (GPU) together with a CPU to accelerate application.
- CPU consists of a few cores optimized for sequential serial processing.
- GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously.
- GPU can be seen as a co-processor of the CPU.
GPU computing

- Uses standard video cards by Nvidia or sometimes ATI.
- Uses a standard PC with Linux, MSW or MacOS.
- Programming model SIMD (Single Instruction, Multiple Data).
- Parallelisation inside card is done through threads.
- SIMT (Single Instruction, Multiple Threads).
- Dedicated software to access the card and start kernels.
- CUDA by Nvidia and OpenCL are the most popular solutions.
GPU computing - Advantages

- Hardware is cheap compared with workstations or supercomputers.
- Simple GPU already inside many desktops without extra investments.
- Capable of thousands of parallel threads on a single GPU card.
- Very fast for algorithms that can be efficiently parallelised.
- Better speedup than MPI for many threads due to shared memory.
- Several new high level libraries hiding complexity: BLAS, FFTW, SPARSE, ...
- In progress.
GPU computing - Disadvantages

- Limited amount of memory available (max. 2-24 GByte).
- Memory transfers between host and graphics card cost extra time.
- Fast double precision GPUs still quite expensive.
- Slow for algorithms without enough data parallelism.
- Debugging code on GPU can be complicated.
- Combining more GPUs to build a cluster is (was?) complex (often done with pthreads, MPI or OpenMP).
- In progress.
GPU computing

Theoretical GFLOP/s

- **NVIDIA GPU Single Precision**
- **NVIDIA GPU Double Precision**
- **Intel CPU Double Precision**
- **Intel CPU Single Precision**

Devices:
- GeForce 780 Ti
- GeForce GTX TITAN
- GeForce GTX 680
- Tesla K20X
- Tesla K40
- Tesla M2090
- Tesla C2050
- Sandy Bridge
- Ivy Bridge
- Westmere
- Bloomfield
- Harpertown
- Woodcrest
- GeForce 8800 GTX
- GeForce GTX 280
- GeForce GTX 480
- GeForce GTX 580
- GeForce 7800 GTX
- GeForce 6800 Ultra
- Pentagon 4
- GeForce FX 5800

Dates:
- Apr-01
- Sep-02
- Jan-04
- May-05
- Oct-06
- Feb-08
- Jul-09
- Nov-10
- Apr-12
- Aug-13
- Dec-14
GPU hardware structure
1. Python tools for parallel computing
2. Parallel Python
   - What is PP?
   - API
3. MPI for Python
   - MPI
   - mpi4py
4. GPU computing with Python
   - GPU computing
   - CUDA
   - PyCUDA
   - Anaconda Accelerate - Numbapro
CUDA

- *Compute Unified Device Architecture* and is a software toolkit by Nvidia.
- Eases the use of Nvidia graphics cards for scientific programming.
- Special C compiler to build code both for CPU and GPU (nvcc).
- C Language extensions: distinguish CPU and GPU functions, access different types of memory on the GPU, specify how code should be parallelized on the GPU, ...
- Library routines for memory transfer between CPU and GPU.
- Extra BLAS, Sparse and FFT libraries for easy porting existing code.
- Mainly standard C on the GPU.
CUDA concepts

- Kernels: special functions executed in parallel on GPU.
- Memory transfer: copy the data between CPU and GPU memories.
- Host = CPU and Device = GPU.
- Thread: processes executed in parallel.
- Blocks: equal-size groups of threads.
- Grid: group of blocks. Execute the kernels.
CUDA programming model

C Program
Sequential
Execution

Parallel kernel
Kernel0<<<>>>() { 

Serial code

Device

Grid 0

Block (0, 0) Block (1, 0) Block (2, 0)

Block (0, 1) Block (1, 1) Block (2, 1)

Serial code

Device

Grid 1

Block (0, 0) Block (1, 0)

Block (0, 1) Block (1, 1)

Block (0, 2) Block (1, 2)
CUDA programming model

Grid

Block (0, 0)  Block (1, 0)  Block (2, 0)

Block (0, 1)  Block (1, 1)  Block (2, 1)

Block (1, 1)

Thread (0, 0)  Thread (1, 0)  Thread (2, 0)  Thread (3, 0)

Thread (0, 1)  Thread (1, 1)  Thread (2, 1)  Thread (3, 1)

Thread (0, 2)  Thread (1, 2)  Thread (2, 2)  Thread (3, 2)
Python tools for parallel computing

Parallel Python
- What is PP?
- API

MPI for Python
- MPI
- mpi4py

GPU computing with Python
- GPU computing
- CUDA
- PyCUDA
- Anaconda Accelerate - NumbaPro
PyCUDA

- Wrapper of Nvidia CUDA for Python.
- Abstractions like `pycuda.driver.SourceModule` and `pycuda.gpucarray.GPUArray` make CUDA programming easier.
- PyCUDA puts the full power of CUDAs driver API at your disposal.
- Automatic Error Checking: All CUDA errors are automatically translated into Python exceptions.
- Speed: PyCUDA’s base layer is written in C++.
- It is necessary to know C-like language.
- Open source: [http://mathema.tician.de/software/pycuda/](http://mathema.tician.de/software/pycuda/)
PyCUDA - Examples

- First example: `pycuda_sumarrays.py`
- More involved example: `pycuda_montecarlo_pi.py`
- GPUArray example: `pycuda_montecarlo_pi_GPUArray.py`
Python tools for parallel computing

Parallel Python
   What is PP?
   API

MPI for Python
   MPI
   mpi4py

GPU computing with Python
   GPU computing
   CUDA
   PyCUDA
   Anaconda Accelerate - Numbapro
Anaconda Accelerate

- Allow developers to rapidly create optimized code that integrates well with NumPy.
- Offers developers the ability to code Python parallel implementations for multicore and GPU architectures.
- [http://docs.continuum.io/accelerate/index.html](http://docs.continuum.io/accelerate/index.html)
- But...it is not free....
- But...Anaconda Academic License.
Numbapro - Features

- Just-in-time compilation to target CPU, Multi CPU or GPU.
- Universal functions (*ufuncs*) and generalized universal functions (*gufuncs*).
- *ufuncs* and *gufuncs* are also compiled on the fly.
- Portable data-parallel programming.
- CUDA-based API is provided for writing CUDA code specifically in Python.
- Bindings to CUDA libraries: cuRAND, cuBLAS, cuFFT.
- [http://docs.continuum.io/numbapro/](http://docs.continuum.io/numbapro/)
Numbapro - Examples

- **ufuncs** example: numbapro_sumarrays.py
- **Just-in-time** example: numbapro_sumarrays_jit.py
- **ufuncs vs. Just-in-time** example: numbapro_saxpy.py
- Target comparision example: numbapro_discriminat.py