Λογισμικό και Προγραμματισμός Συστημάτων Υψηλής Επίδοσης

«HPC and Task Parallelism in Python»

Παναγιώτης Χατζηδούκας, Ευάγγελος Δερματάς*

Introduction

- Python is an interpreted language
  - no pre-compiled binaries, all code is translated on-the-fly to machine instructions
  - byte-code as a middle step which may be stored (.pyc)
- All objects are dynamic in Python
  - nothing is fixed == optimisation nightmare
  - lot of overhead from metadata
- Flexibility is good but comes with a cost!

Performance and Parallelization

- Performance improvement techniques
  - Array based computations with NumPy
  - Using extended Cython programming language
  - Embed compiled code in a Python program (C/C++, Fortran)
  - Utilize parallel processing
- Parallelisation strategies for Python
  - Global Interpreter Lock (GIL)
    - CPython's memory management is not thread-safe
    - no threads possible, except for I/O etc.
    - affects overall performance of threading
  - Process-based "threading" with multiprocessing
    - fork independent processes that have a limited way to communicate
  - Message-passing is the Way to Go to achieve true parallelism in Python

Multiprocessing

- Underlying OS used to spawn new independent subprocesses
- processes are independent and execute code in an asynchronous manner
  - no guarantee on the order of execution
- Communication possible only through dedicated, shared communication channels: must be created before a new process is forked
  - shared memory, data manager
  - pipes: direct communication between two processes
  - queues: work sharing among a group of processes
  - pool of workers: offloading tasks to a group of worker processes

Multiprocessing example I

```
from multiprocessing import Process
import os

def hello(name):
  print("Hello", name)
  print("My PID is", os.getpid())
  print("My parent's PID is", os.getppid())

p = Process(target=hello, args=('Alice', ))
p.start()
print('Spawned a new process from PID', os.getpid())
p.join()
```

Multiprocessing example II

```
from multiprocessing import Process

def squared(x):
    print(x**2)

# create parallel processes
procs = [Process(target=squared, args=[x]) for x in range(10)]

# start all processes
for p in procs:
    p.start()

# wait for all tasks to finish
for p in procs:
    p.join()
```

Pool of workers

```
from multiprocessing import Pool
import time

def f(x): return x**2

pool = Pool(4)

# blocking alternative
result = pool.map(f, range(10))
print(result)

# non-blocking alternative
result = pool.map_async(f, range(10))
while not result.ready():
  time.sleep(1)
print(result.get())
```

Multiprocessing and shared memory

```
from multiprocessing import Process, Array

def squared(a, i):
    a.acquire()
    a[i] = i * i
    a.release()

numbers = Array('i', range(10))
p = [Process(target=squared, args=[numbers, i]) for i in range(10)]

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

print(numbers[:])
```

mpi4py

- Communication between processes
  - sending and receiving messages between two processes
  - sending and receiving messages between several processes
- Synchronization between processes
- Communicator creation and manipulation
- Advanced features (e.g. user defined datatypes, one-sided communication and parallel I/O)

- Installation and usage
  - conda install mpi4py
  - mpi-hello.py: from mpi4py import MPI
  - mpiexec -n 4 python mpi-hello.py

MPI datatypes

- MPI has a number of predefined datatypes to represent data
  - e.g. MPI.INT for integer and MPI.DOUBLE for float
- No need to specify the datatype for Python objects or Numpy arrays
  - objects are serialised as byte streams
  - automatic detection for NumPy arrays
- If needed, one can also define custom datatypes
  - for example to use non-contiguous data buffers

- Possible to communicate arbitrary Python objects
- NumPy arrays can be communicated with nearly same speed as in C/Fortran

mpi4py

```
# mpi-hello.py
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
print("I am rank {} in group of {} processes.".format(rank, size))
```

```
# mpi-send.py
from mpi4py import MPI

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

if rank == 0:
   data = {'a': 7, 'b': 3.14}
   comm.send(data, dest=1)
elif rank == 1:
   data = comm.recv(source=0)

if rank == 1:
   print("Received: " + str(data))
```

Task parallelism

- Main form of code parallelization
- Launching and distributing asynchronous tasks across different processors
  - of a local machine or on a cluster or supercomputer
- The tasks can be
  - completely decoupled: a set of independent jobs
  - part of an algorithm: the task results are processed and drive the next step
- Common use cases
  - Applying the same function to different data
  - Parametric searches
  - Numerical optimization and uncertainty quantification

Task parallelism in Python, sequential case

- Executing the same function on a list of input data:

```
results = []
for x in data:
  results.append(work(x))
```

- Using the Python built-in map() function:

```
results = list(map(work, data))
```

- Questions: How can we
  - parallelize the above cases on both shared and distributed memory platforms?
  - provide lightweight support for nested loops and map functions?
  - take advantage of MPI (performance vs fault-tolerance)?
  - have all the above in a simple python package?

Task parallelism in Python

Platform	Package	Features
Single node	multiprocessing, futures	Task-pool for processes - ProcessPool, ThreadPool
Distributed/ MPI-based	mpi4py.futures [1]	MPI extension of concurrent.futures Master-worker execution model
	dtm [2]	Master participates, partial support of nested tasks Issues with MPI thread-safety, discontinued-replaced by scoop
Distributed/ Not MPI	scoop [3]	support of nested tasks (coroutines), communication using sockets or message-queue (broker), global data through containers
	celery [4]	message-queue based, client and task codes are decoupled, partial support for nested tasks (no wait)
	dask [5]	client-scheduler-worker approach, nesting requires additional workers
	pycompss [6]	Compiler-based approach, java runtime, many software dependencies

1. https://mpi4py.readthedocs.io/en/stable/mpi4py.futures.html
2. http://deap.gel.ulaval.ca/doc/0.9/api/dtm.html
3. http://pyscoop.org (ACM XSEDE 2013)
4. http://www.celeryproject.org/
5. http://dask.pydata.org/en/latest/
6. https://pypi.org/project/pycompss/ (IJHPCA 2017)

Using the multiprocessing module

- Submit tasks with apply_async()

```
from multiprocessing import Pool
p = Pool(4)

tasks = []
for i in data:
  t = p.apply_async(work, (i,))
  tasks.append(t)

for t in tasks:
  print(t.get())
```

- Submit tasks with map

```
from multiprocessing import Pool
p = Pool(4)
results = list(p.map(work, data))
```

Using the concurrent.futures package

- PEP 3148: futures - execute computations asynchronously
- Using the ProcessPoolExecutor and spawning tasks with submit()

```
from multiprocessing import Pool
import concurrent.futures

p = concurrent.futures.ProcessPoolExecutor(max_workers=4)
tasks = []
for i in data:
  tasks.append(p.submit(work, i))
concurrent.futures.wait(tasks)

for t in tasks:
  print(t.result())
```

- Using the map() function

```
p = concurrent.futures.ProcessPoolExecutor(max_workers=4)
results = list(p.map(work, data))
```

- ProcessPoolExecutor ←→ ThreadPoolExecutor

Using the mpi4py.futures package

- Similar interface to the concurrent.futures package (submit/wait and map)
- Using the MPIPoolExecutor from the mpi4py.futures package

```
from multiprocessing import Pool
from mpi4py.futures import MPIPoolExecutor
if __name__ == '__main__':
  p = MPIPoolExecutor()
  if p is not None:
    data = list(range(10))
    results = list(p.map(work, data))
```

- N+1 processes needed for 1 master + N workers

```
mpirun -n 5 python3 -m mpi4py.futures demo.py
```

- Asymmetry: the master process does not participate in the execution of the tasks

Using the dtm framework (deap 0.9.2)

- Task spawning can be done with apply_async/waitAll or map
- Using map

```
from deap import dtm
def main():
  data = range(10)
  results = dtm.map(work, data)

if __name__ == '__main__':
  dtm.start(main)
```

- Use mpirun to start 4 workers

```
mpirun -n 4 python3 demo.py
```

Using the scoop framework

- Task spawning can be done with the futures API (submit/wait or map)
- Using map

```
from scoop import futures
if __name__ == "__main__":
  data=range(10)
  results = list(futures.map(work, data))
```

- Execute with 4 workers

```
python3 -m scoop -n 4 demo.py
```

Using the celery framework

- tasks.py

```
from scoop import futures
from celery import Celery
@app.task
def work(input):
  ...
  return output
```

- master.py

```
from celery import group
from tasks import work
job = group(work.s(i) for i in range(10))
task_group = job.apply_async()
results=task_group.join()
```

- Start the broker

```
rabbitmq-server -detached
```

- Start the celery workers (tasks.py)

```
celery -A tasks worker --concurrency=2
```

- Run the client code

```
python3 master.py
```

Using the dask distributed framework

- demo.py

```
from distributed import Client
from dask.distributed import wait

def work(input):
  ...
  return output

client = Client('127.0.0.1:8786’)
L = client.map(hello, range(10))
wait(L)
results = client.gather(L))
```

- Start the scheduler

```
dask-scheduler &
```

- Start the workers

```
dask-worker --nprocs 4 --nthreads 1 \
9.4.222.129:8786 &
```

- Start the client code

```
python3 demo.py
```

Summary

Framework	Clusters	Nested tasks	MPI	
multiprocessing, futures (PEP 3184)	No	No	No	
mpi4py.futures (PEP 3184)	Yes	No	Yes	Asymmetric master-worker
dtm	Yes	Not efficiently (threads)	Yes	Thread-safety issues, discontinued
scoop (PEP 3184)	Yes	Yes, coroutines	No	Optional MPI-based launching, single message-queue, module for arbitrary sharing of constants objects
celery	Yes	No	No	Decoupled code for tasks and client, service oriented
dask	Yes	Not efficiently (extra workers)	No	Master-scheduler-worker model
pycompss	Yes	Yes	No	Java runtime
torcpy (~PEP 3184)	Yes	Yes	Yes	Distributed queues, control of worker placement, MPI SPMD code

The torcpy tasking library

- Pure Python tasking library for shared and distributed memory systems
  - Originates from TORC, a C/C++ tasking library for HPC clusters
  - Built on top of MPI (mpi4py) and threads
- Main features
  - Transparent data transfer and load-balancing
  - Distributed - multilevel queues
  - Nested parallelism
  - Orthogonal to MPI code
  - Portable, extensible

torcpy API (~PEP 3148)

- **`torc.submit(f, *args, qid=-1, callback=None)`**
  - **`f`**: task function
  - **`args`**: arguments
  - **`qid`**: MPI rank where the task is submitted for execution, -1: cyclic distribution
  - **`callback`**: function to be called on the rank that spawned the task, upon its completion
- **`torc.wait()`**:
  - wait for all child tasks to finish
  - The underlying worker thread is released and can execute other tasks
- **`torc.map(f, chunksize, *seq)`**:
  - execute function on a sequence (list) of arguments
  - Returns a list with the results of all tasks
  - Similar to the map() function of python and other packages
  - Straightforward usage of existing examples

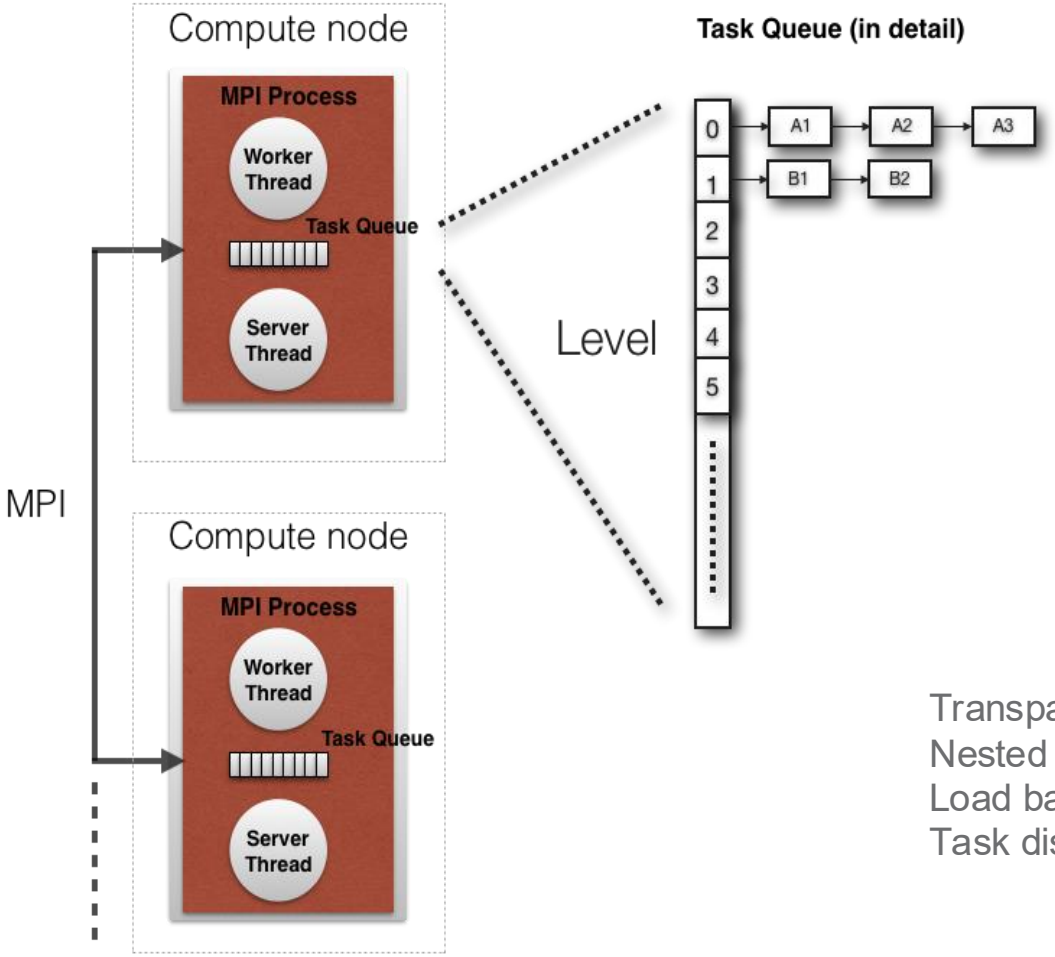
Additional API

- **`torc.spmd(f, *args)`:**
  - execute f() on all MPI ranks
  - Legacy MPI code can be used within f()
- **`torc.start(f)`:**
  - called from __main__, launches primary task f() on rank 0
- **`torc.self()`:**
  - returns the descriptor of the currently executed task
- **`torc.worker_id()`, `torc.num_workers()`:**
  - Global worker thread id and number of workers
- **`torc.node_id()`, `torc.num_nodes()`:**
  - MPI rank and size
- **`torc.enable_stealing()`, `torc.disable_stealing()`:**
  - control internode task stealing

Pure Python3 implementation

- Third-party packages: mpi4py, threading, queue
- Tasks are instantiated as dictionaries (faster than objects)
- The result of the task function is transparently stored in the task descriptor (future) on the rank that spawned the task
  - Can be accessed as task.result()
  - Similarly, the input parameters can be accessed as task.input()
- Remote operations are performed asynchronously through a server thread
  - task insertion in the local queue
  - reception of results
  - task stealing request

Runtime architecture



Transparent data movement
Nested parallelism
Load balancing through task stealing
Task distribution schemes

Example 1

- Using the submit/wait methods

```
import torc

def main():
  data = range(10)
  tasks = []
  for d in data:
    tasks.append(torc.submit(work, d))
  torc.wait()
  for t in tasks:
    print(t.result())

if __name__ == '__main__':
  torc.start(main)
```

- Use MPI to start 4 workers

```
mpirun -n 4 python3 example1.py
```

Example 2

- Using the map() function

```
import torc

def main():
  data = range(10)
  results = torc.map(work, data)
  print(results)

if __name__ == '__main__':
  torc.start(main)
```

Example 3 – recursive fibonacci

```
def fib(n):
  if n == 0 or n == 1:
    result = n
  else:
    n_1=n-1
    n_2=n-2
    if n < 30:
      result1=fib(n_1)
      result2=fib(n_2)

      result=result1+result2
    else:
      task1 = torc.submit(fib, n_1)
      task2 = torc.submit(fib, n_2)
      torc.wait()

      result = task1.result()
              +task2.result()

  return result
```

```
def main():
  n = 35
  result=fib(n)
  print("fib({})={}".format(n, result))
```

Example 4 – map for CMA

```
import cma  # python package for numerical optimization
import torc

def rosenbrock(x):
  """Rosenbrock test objective function"""
  # ...
  return result

def main():
    es = cma.CMAEvolutionStrategy(2 * [0], 0.5, {'popsize':8})
    while not es.stop():
        solutions = es.ask()
        es.tell(solutions, torc.map(rosenbrock, solutions))
        es.logger.add(es)  # write data to disc to be plotted
        es.disp()

cma.plot()
```

Example 5 – reduction on rank 0 using callbacks

```
sum = 0

def cb(arg):
  global sum
  sum = sum + arg

def work(x):
  y=x**2
  return y

def main():
  data = range(10)

  tasks = []
  for d in data:
    t = torc.submit(taskfun, d, callback=cb)
    tasks.append(t)

  torc.wait()

  print("Sum=", sum)
```

Example 6 – calling SPMD code

```
from mpi4py import MPI

N = 3
A = numpy.zeros(N, dtype=numpy.float64)

def bcast_task(root):
  global A
  comm = MPI.COMM_WORLD
  comm.Bcast([A, MPI.DOUBLE], root=root)  # Broadcast A from rank 0

def main():
  global A
  A = 100*numpy.arange(N, dtype=numpy.float64)
  torc.spmd(bcast_task, 0)
  ...
```

Tuning and benchmarking

- TORC, the C/C++ counterpart of torc_py has been used extensively on small and large scale HPC environments
  - Euler cluster (ETH), Piz Daint (CSCS)
- Performance tuning with existing benchmark codes for mpi4py.futures (julia, primes)
- Main issues:
  - Avoid blocking MPI_Recv calls, yield often to allow multiple Python threads to progress
  - This works well for one worker thread and the server thread
  - For multiple worker threads, the performance can be affected by the Python GIL

Application case: data preprocessing

- Typical preprocessing stage in Deep Learning workloads
  - Transform datasets of raw images to a single file in HDF5 format
- For each image
  - Open file and read image data (includes decompression for JPEG format)
  - Resize and rescale image (possibly also apply some preprocessing filters)
  - Write result in HDF5 file

```
def process_train_image(i, target_dim):
  global reader, pipe

  im, label = reader.get_train_image(i)
  im = pipe.filter(im)

  dx = image_to_4d_tensor(im, target_dim)
  dy = label

  return dx, dy
```

```
sequence_i = range(n_train)
sequence_t = [target_dim] * n_train
task_results = torc.map(
                process_train_image,
                chunksize, sequence_i,
                sequence_t)

i = 0
for t in task_results:
  dx, dy = t
  dataset_x[i, :, :, :] = dx
  dataset_y[i] = dy
  i = i+1
```

Application case: data preprocessing

- IBM Power 8 node: 20 cores with 8 threads each
- Imagenet: very large dataset of 1000 classes

