COMP S362F Unit 13 High-performance computing

Kendrew Lau

Contents

- Python performance
- Message passing interface (MPI)
- Python profiling

Python performance

Python is slow but good!

- Python is slow, but it is not a problem in most cases
- As a “glue” or scripting language, often invoking heavy duties that are implemented in non-Python ways
    - I/O operations, networking, file access
    - Accessing services of other systems
    - Executing native code for computation
- Easy to learn and use, fast to write (relatively)

Improving Python performance

- Buy better hardware
- Adopt appropriate algorithms, language features, and libraries
- Use libraries containing native code for heavy jobs
- Create modules of native code for critical parts of applications
- Consider alternative Python implementations
- Employ concurrency

Libraries containing native code

- Implementing specific algorithms and data structures in low-level languages (e.g. C/C++) achieves very good performance
- E.g. NumPy, https://numpy.org/
  - A library of arrays and mathematical functions
  - Used in many other scientific libraries
  - Performance similar to coding in C/C++
- E.g. TensorFlow, https://www.tensorflow.org/
  - A library of deep learning and neutral networks
  - Different versions for various hardware, e.g. CPU, GPU, etc

Creating native code modules

- Native code usually means C/C++, requiring a compiler
- Write C/C++ code directly
    - Write all code yourself
    - Use SWIG, http://www.swig.org/, to generate interface code
- Cython, https://cython.org/
    - Write in a Python-like language or pure Python, and compile to native code
- Numba, https://github.com/numba/numba
    - Apply a decorator to Python code, and compile to native code automatically

A taste of Cython

```python
# Python version
def fib(n):
    a, b = 0.0, 1.0
    for i in range(n):
        a, b = a + b, a
    return a
```

```cython
# Cython version
def fib(int n):
    cdef int i
    cdef double a=0.0, b=1.0
    for i in range(n):
        a, b = a + b, a
    return a
```

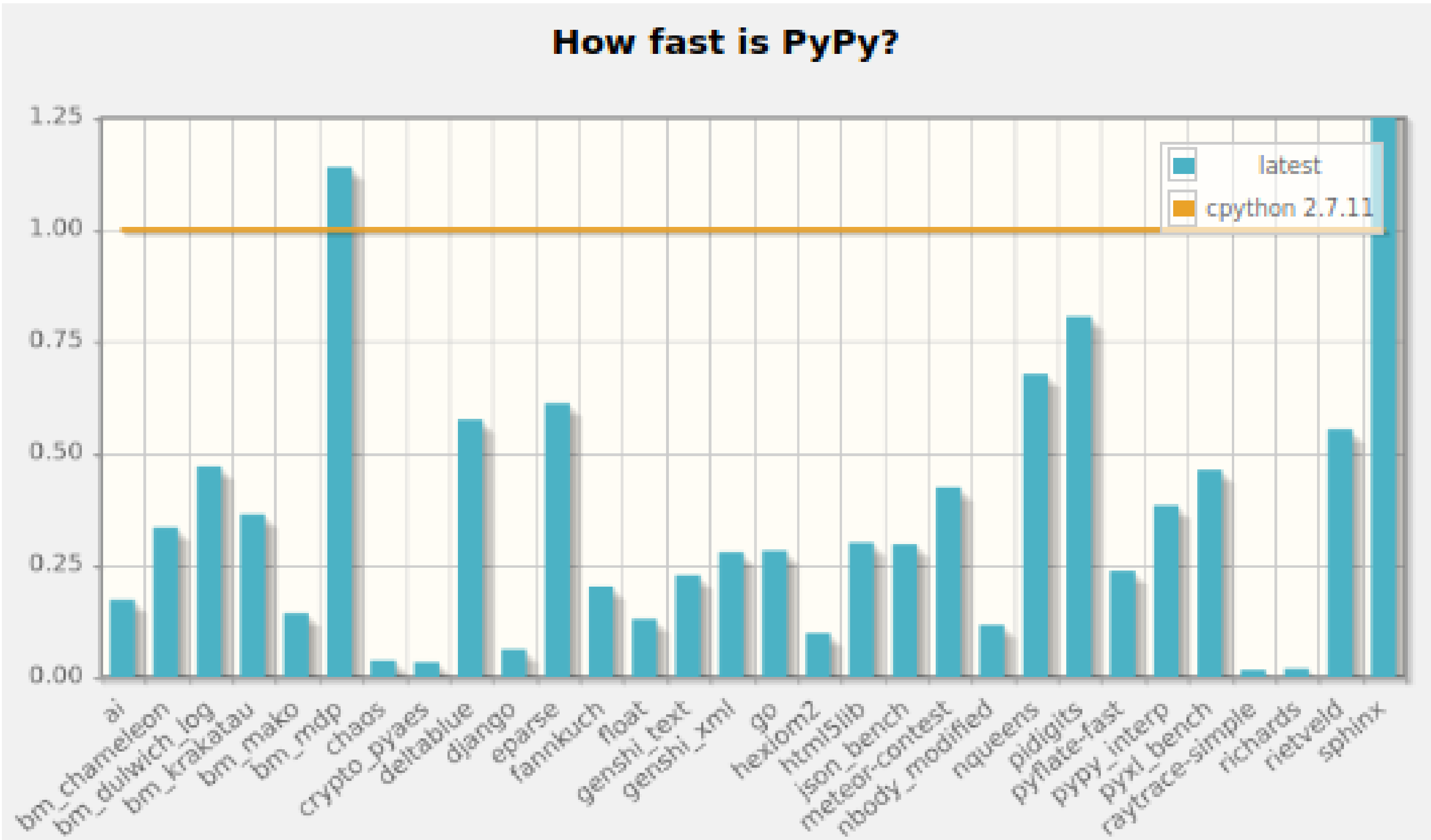
A taste of Numba

```python
from numba import jit
import numpy as np

x = np.arange(100).reshape(10, 10)

# Set "nopython" mode for best performance, equivalent to @njit
# Function is compiled to machine code when called the first time
@jit(nopython=True)
def go_fast(a):
    trace = 0.0
    for i in range(a.shape[0]):   # Numba likes loops
        trace += np.tanh(a[i, i]) # Numba likes NumPy functions
    return a + trace              # Numba likes NumPy broadcasting

print(go_fast(x))
```

Python implementations

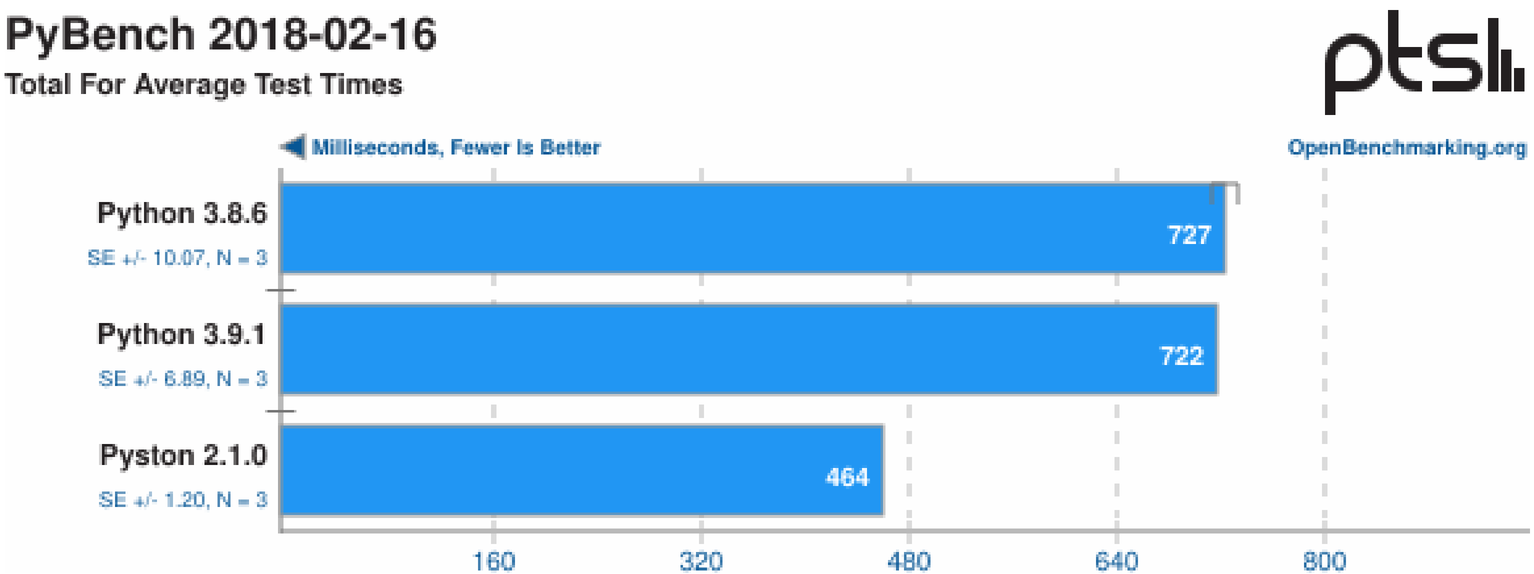
- CPython, https://www.python.org/, reference implementation
- PyPy, https://www.pypy.org/, JIT
- Jython, https://www.jython.org/, the JVM
- IronPython, https://ironpython.net/, the .NET
- Cinder, https://github.com/facebookincubator/cinder, JIT
- Pyston, https://www.pyston.org/, JIT

PyPy performance



Source: https://www.pypy.org/

Pyston performance



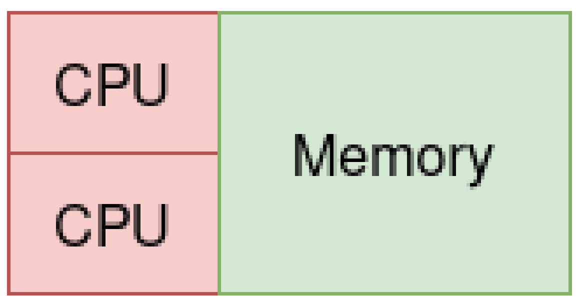
Source: https://www.phoronix.com/scan.php?page=news_item&px=Pyston-2.1-vs-Python-3.8-3.9

Concurrency

- Use threads, processes, and asynchronous programming
  - Built-in modules `threading`, `multiprocessing`, `concurrent.futures`, `asyncio`
- High-performance computing technologies and standards
  - *Message Passing Interface (MPI)*
    - A standard for passing messages in distributed-memory systems
  - *Open Multi-Processing (OpenMP)*
    - An API for shared-memory multiprocessing
  - *General Purpose Graphic Processing Unit (GPGPU)* programming
    - *Compute Unified Device Architecture (CUDA)*, NVIDIA proprietary
    - *Open Computing Language (OpenCL)*, open standard

OpenML

- *Open Multi-Processing (OpenMP)*, https://www.openmp.org/
- An API for shared-memory multiprocessing
  - Essentially multithreading, executing tasks concurrently using threads
  - C, C++, FORTRAN
- Three primary components
  - Compiler directives
  - Runtime libraries
  - Environment variables



A taste of OpenML

- Very simple way to do multithreading
  - Insert OpenMP parallel directives to create tasks for threads
  - But updating shared variables need special treatment

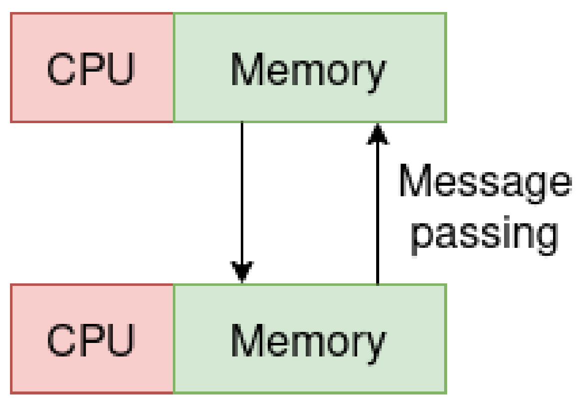
```
// repetitive work: OK
#pragma omp parallel for
for (i=0; i<N; i++)
  a[i] = b[i] + c[i];
```

```
// repetitive update: NG
#pragma omp parallel for
for (i=0; i<N; i++)
  sum = sum + b[i] * c[i];
```

Message Passing Interface (MPI)

MPI

- *Message Passing Interface (MPI)*, https://www.mpi-forum.org/
- A standard for passing messages in distributed-memory systems
    - Work on individual computers and clusters
- Many implementations available
    - OpenMPI, MPICH, HP-MPI, MS-MPI
- *Single Program, Multiple Data (SPMD)*
    - Multiple processes execute the same source code, each having its own data



Installing MPI

- MPI implementation
    - Windows: Microsoft MPI
        - Install `msmpisetup.exe`, from https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi
        - `set PATH=%PATH%;"C:\Program Files\Microsoft MPI\bin"`
    - Mac/Linux: install package `openmpi` or the like
- The MPI for Python package, `mpi4py`
    - Install: `pip install mpi4py`
    - Execute program: `mpiexec [-n n] python program.py`

Programming MPI

- *Single Program, Multiple Data (SPMD)*
  - One source code is executed by multiple processes, each having its own data and logic
- In MPI, each process has its own ID, called *rank*
  - By convention, the process with rank 0 is the *root*
- The source code has conditional execution
  - Each process executes some logic based on its rank
  - Processes can communicate with each other

Essential mpi4py API

```
from mpi4py import MPI

comm = MPI.COMM_WORLD  # the world intracommunicator
rank = comm.Get_rank() # the rank of current process
size = comm.Get_size() # the number of processes

comm.send(data, dest=1)    # send data to rank 1
data = comm.recv(source=0) # receive data from rank 0

data = comm.bcast(data, root=0) # rank 0 sends, others receive
```

```
send(obj, dest, tag=0)
recv(buf=None, source=ANY_SOURCE, tag=ANY_TAG, status=None)
bcast(obj, root=0)
```

Hello MPI example

```python
# File: hello.py
import os
from mpi4py import MPI

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

print("Rank:", rank, " Size:", size, " PID:", os.getpid())
```

```
$ mpiexec python hello.py
Rank: 0  Size: 4  PID: 841055
Rank: 2  Size: 4  PID: 841057
Rank: 1  Size: 4  PID: 841056
Rank: 3  Size: 4  PID: 841058
```

Sending and receiving data example

```python
# File: send_recv.py
import random
from mpi4py import MPI

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

if rank == 0:
    data = random.randint(1, 100)
    for dest in range(1, size):
        comm.send(data, dest)
    print(f"Rank {rank} sends {data}")
else:
    data = comm.recv()
    print(f"Rank {rank} receives {data}")
```

```
$ mpiexec python send_recv.py
Rank 0 sends 45
Rank 1 receives 45
Rank 2 receives 45
Rank 3 receives 45
```

Broadcasting data example

```python
# File: bcast.py
import random
from mpi4py import MPI

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

if rank == 0:
    data = random.randint(1, 100)
else:
    data = None

data = comm.bcast(data, root=0)

if rank == 0:
    print(f"Rank {rank} broadcasts {data}")
else:
    print(f"Rank {rank} receives {data}")
```

```
$ mpiexec python bcast.py
Rank 0 broadcasts 20
Rank 1 receives 20
Rank 2 receives 20
Rank 3 receives 20
```

Summation example

```python
# File: summation.py
import timeit
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
```

Summation example (cont.)

```python
# File: summation.py (cont.)
if rank == 0:
    N = int(1e8)
    start_time = timeit.default_timer()
    part_size = N // (size - 1)
    for part in range(size - 1):
        start = part * part_size
        stop = (part + 1) * part_size if part != size - 2 else N + 1
        comm.send((start, stop), dest=part+1)
    total = 0
    for r in range(1, size):
        total += comm.recv(source=r)
    end_time = timeit.default_timer()
    print(f"{total} {end_time - start_time}s")
else:
    start, end = comm.recv()
    total = sum(range(start, end))
    comm.send(total, dest=0)
```

Summation example (cont.)

```
$ mpiexec -n 4 python summation.py
5000000050000000 0.408634713036082s

$ mpiexec -n 3 python summation.py
5000000050000000 0.594822603976354s

$ mpiexec -n 2 python summation.py
5000000050000000 1.066621191974263 6s
```

Python profiling

Python profiling

- A *profile* is a set of statistics that describes how often and for how long various parts of the program executed. (Python doc)
- In software optimization, identify bottlenecks by profiling
  - Do not guess or improve randomly!
  - Write a correct functional program, before optimizing it
- Python supplies two profilers
  - `cProfile`: a C extension, low overhead, recommended
  - `profile`: a pure Python module, high overhead
- Usage: `cProfile.run(command, filename=None, sort=-1)`

Profile report format

```
ncalls tottime percall cumtime percall filename:lineno(function)
     1   0.000   0.000   1.424   1.424 <string>:1(<module>)
     1   0.423   0.423   1.424   1.424 profile1.py:5(test_fn)
     2   1.001   0.501   1.001   0.501 {built-in method time.sleep}
```

- `ncalls`: number of calls
- `tottime`: total time spent in the given function, excluding time for calling sub-functions
- `percall`: quotient of `tottime` divided by `ncalls`
- `cumtime`: cumulative time spent in this and all sub-functions
- `percall`: quotient of `cumtime` divided by primitive calls
- `filename:lineno(function)`: file name, line number, function name

Profiling example 1

```
# File: profile1.py
import cProfile, time

N = int(1e7)

def test_fn():
    time.sleep(0.5)
    total = 0
    for i in range(N+1):
        total += i
    print(total)
    time.sleep(0.5)

cProfile.run("test_fn()", sort="cumtime")
```

Profiling example 1 (cont.)

```
# Output:
```

```
50000005000000
      7 function calls in 1.424 seconds

Ordered by: cumulative time

ncalls tottime percall cumtime percall filename:lineno(function)
     1   0.000   0.000   1.424   1.424 {built-in method builtins.exec}
     1   0.000   0.000   1.424   1.424 <string>:1(<module>)
     1   0.423   0.423   1.424   1.424 profile1.py:5(test_fn)
     2   1.001   0.501   1.001   0.501 {built-in method time.sleep}
     1   0.000   0.000   0.000   0.000 {built-in method builtins.print}
     1   0.000   0.000   0.000   0.000 {method 'disable' of
'_lsprof.Profiler' objects}
```

Profiling example 2

```python
# File: profile2.py
import cProfile, multiprocessing, threading

N = int(1e7)
n = 0

def inc_no_lock():
    global n
    n = 0
    for i in range(N):
        n += 1
    print(n)
```

Profiling example 2 (cont.)

```python
# File: profile2.py (cont.)
def inc_process_lock():
    global n
    n = 0
    lock = multiprocessing.Lock()
    for i in range(N):
        with lock:
            n += 1
    print(n)

def inc_thread_lock():
    global n
    n = 0
    lock = threading.Lock()
    for i in range(N):
        with lock:
            n += 1
    print(n)

cProfile.run("inc_no_lock(); inc_process_lock(); inc_thread_lock()",
             sort="cumtime")
```

Profiling example 2 (cont.)

```
# Output:
```

```
10000000
10000000
10000000
      50004554 function calls (50004422 primitive calls) in 11.227
seconds

Ordered by: cumulative time

  ncalls tottime percall cumtime percall filename:lineno(function)
    13/1   0.000   0.000  11.227  11.227 {built-in method
builtins.exec}
       1   0.000   0.000  11.227  11.227 <string>:1(<module>)
       1   3.448   3.448   7.756   7.756
profile2.py:13(inc_process_lock)
       1   2.326   2.326   2.918   2.918
profile2.py:22(inc_thread_lock)
10000000   1.718   0.000   2.248   0.000 synchronize.py:97(__exit__)
10000000   1.526   0.000   2.053   0.000 synchronize.py:94(__enter__)
10000046   0.592   0.000   0.592   0.000 {method '__exit__' of
'_thread.lock' objects}
       1   0.553   0.553   0.553   0.553 profile2.py:6(inc_no_lock)
10000000   0.530   0.000   0.530   0.000 {method '__exit__' of
'_multiprocessing.SemLock' objects}
10000000   0.527   0.000   0.527   0.000 {method '__enter__' of
'_multiprocessing.SemLock' objects}
```

The end