# Introduction to Algorithms in Python

Juha Jäykkä

January 11, 2017

## Contents



# 1 Efficient and portable I/O

- You will eventually want to read and write data, too
- **Badly implemented I/O can easily take 99% of your run-time!**

## 1.1 Portable I/O

- plain text is portable, but
  - horribly space wasting (unless compressed, which then harms portability)
  - horribly slow as everything needs to be converted
    - unless your actual calculations are done using letters
  - cannot really be parallelised so in parallel even more slower
- some file formats will be hard or impossible to open on another computer so
- avoid machine dependent I/O, like
  - The unformatted IO in Fortran
    - unlikely to be efficient
    - likely to be very hard to open on another machine or Fortran library
  - C's `fwrite(data, size, count, file)`
    - with the right `size` and `count` this can be very efficient for a single-threaded application
    - but not very portable: few data types are guaranteed to be equivalent across machines and byte-order can also change
- some good libraries to portable I/O
  - HDF5 is de facto high-performance data format and very standardised

    * cross-language, cross-platform: **available in every imaginable system and language**!
    * will **not** load the file to memory unless it is sliced or altered: useful for processing files which are too large for your memory if you do not need to alter the data
  - netcdf4 is also quite efficient (in fact it's HDF5 underneath)
  - sometimes numpy's `save`, `savez` and `savez_compressed` are ok
    * remember to pass `allow_pickle=False` to maintain portability
    * when loading, pass `mmap_mode` parameter to use memmaped IO: useful for large files when only part of it is needed
    * but have a look at Blaze, too

### 1.1.1 numpy can import almost any text file easily

- suppose you have a file like this

```
# This file contains a bunch of point particles of varying locationg, speeds,
# masses and charges
X Y Z Vx Vy Vz mass charge
0 0 0 0 0 0.1 100.0 0.0
1.0 0 4 0 10.0 0 10.0 -1.0
0 2 0 0.1 0 0.1 1000.0 0.0
2 2 -3 0.1 0 0.1 100.0 1.0
# A comment line
```

- you can import this with numpy easily

```
import numpy
data = numpy.genfromtxt("files/genfromtxt_example_data.txt", comments="#",
                        delimiter="\t", skip_header=3)
print(data)
```

- please see `help(numpy.genfromtxt)` for full documentation: the function is capable of ingesting almost any kind of textual data, even strings!
- but it is not fast, no text import ever is
- we'll later show how to plot this

## 1.2 Performant I/O

- all the above formats can be internally and transparently compressed, which can help save disc space
  - but compression is not magic as we shall see

### 1.2.1 IO using numpy

```
import numpy
import os
import tempfile
import cProfile
import pstats
data=numpy.random.random((1000,1000,100))
tempfiles = [tempfile.TemporaryFile(dir=".") for i in [0,1,2,3]]
cps = [cProfile.Profile() for i in range(len(tempfiles))]
runs = ["numpy.savez", "numpy.savez_compressed", "numpy.savez_compressed",
        "numpy.savez_compressed"]
for i,r in enumerate(runs):
    if (i==2):
        data[100:900,100:900,30:70]=0.0
    if (i==3):
        data = numpy.ones((1000,1000,100), dtype=numpy.float64)
    cps[i].runcall(eval(r), tempfiles[i], {"array_called_data":data})

print('''Time spent and file sizes:
  uncompressed random data:   {uncompt:g}\t{uncomps}
  compressed random data:     {compt:g}\t{comps}
  compressed semirandom data: {semit:g}\t{semis}
  compressed zeros:           {zerot:g}\t{zeros}'''.format(
      uncomps=os.stat(tempfiles[0].name).st_size,
      comps=os.stat(tempfiles[1].name).st_size,
      semis=os.stat(tempfiles[2].name).st_size,
      zeros=os.stat(tempfiles[3].name).st_size,
      uncompt=pstats.Stats(cps[0]).total_tt,
      compt=pstats.Stats(cps[1]).total_tt,
      semit=pstats.Stats(cps[2]).total_tt,
      zerot=pstats.Stats(cps[3]).total_tt
  ))
```

```
Time spent and file sizes:
  uncompressed random data:   18.5261 1199032850
  compressed random data:     150.646 902644270
  compressed semirandom data: 110.384 680404717
  compressed zeros:           20.7739 1747540
```

- floating point numbers are often almost random from a compression algorithm's point of view
- for random data the breakeven point with my laptop is around 6 MB/s disc write speed: slower than that and compression wins
  - unfair comparison since only one core used for compression and the algorithm used is embarrassingly parallel
  - unfair also because the data comes from a fast local SSD: the write speed is over 50 MB/s

- in supercomputing environments disc write speeds of 5 GB/s are normal, but that would require compression speed to go up by over 1000x or more to make compression worth while
- but this all depends on both the compression factor and time it takes to compress: the last case obviously benefits even with 50 MB/s disc and single-core compression
- **bottom line:** only useful in special cases and when disc-space is tight but CPU seconds are not

### 1.2.2 HDF5 and h5py: writing and transparent compression

- HDF5's `szip` algorithm is supposed to understand floating point numbers and compress smartly
  - unfortunately we do not have it available here
- learn by example: a simple 3D array of random numbers

```
import numpy
import h5py
import os
import tempfile
import cProfile
import pstats

def h5py_create(filename, datadict, compression):
    '''Create a new HDF5 file called "filename" and save the values of
    "datadict" into it using its keys as the dataset names; create an
    attribute called "compression" holding the value of "compression"
    parameter. '''
    f = h5py.File(filename, mode="w")
    attrvalue = "nothing interesting for now"
    f.attrs.create("top-level-attribute", attrvalue,
    dtype="S{x}".format(
        x=len(attrvalue)))
    for name,value in datadict.items():
        ds = f.create_dataset(name, data=value, compression=compression, chunks=True)
        ds.attrs.create("compression", str(compression),
    dtype="S{x}".format(
        x=len(str(compression))))
    return

def szip_available():
    '''Try to create a dataset using szip: return True if succeeds, False
    on ValueError (szip not available) and raise on others.'''
    import tempfile
    tempf = tempfile.NamedTemporaryFile(dir=".")
    f = h5py.File(tempf.name,"w")
    try:
        f.create_dataset("foo", shape=(10,10), dtype="f8", compression="szip")
```

```
    except ValueError:
        ret = False
    else:
        ret = True
    finally:
        f.close()
    return ret

data=numpy.random.random((1000,1000,100))
tempfiles = [tempfile.NamedTemporaryFile(dir=".") for i in [0,1,2,3]]
cps = [cProfile.Profile() for i in range(len(tempfiles))]
if (szip_available()):
    comp="szip"
else:
    comp="gzip"
runs = [None] + 3*[comp]
for i,r in enumerate(runs):
    if (i==2):
        data[100:900,100:900,30:70]=0.0
    if (i==3):
        data = numpy.ones((1000,1000,100), dtype=numpy.float64)
    cps[i].runcall(h5py_create, tempfiles[i].name, {"array_called_data":data}, r)

print('''Time spent writing hdf5 data and file sizes:
  uncompressed random data:   {uncompt:g}\t{uncomps}
  {comp} compressed random data:     {compt:g}\t{comps}
  {comp} compressed semirandom data: {semit:g}\t{semis}
  {comp} compressed zeros:           {zerot:g}\t{zeros}'''.format(
      uncomps=os.stat(tempfiles[0].name).st_size,
      comps=os.stat(tempfiles[1].name).st_size,
      semis=os.stat(tempfiles[2].name).st_size,
      zeros=os.stat(tempfiles[3].name).st_size,
      uncompt=pstats.Stats(cps[0]).total_tt,
      compt=pstats.Stats(cps[1]).total_tt,
      semit=pstats.Stats(cps[2]).total_tt,
      zerot=pstats.Stats(cps[3]).total_tt,
      comp=comp
  ))

Time spent writing hdf5 data and file sizes:
  uncompressed random data:   0.958511 867455344
  gzip compressed random data:     45.6555 756436309
  gzip compressed semirandom data: 30.6861 564465654
  gzip compressed zeros:           7.04421 2177388
```

### 1.2.3 Always write huge chunks of data

- latency is more likely to ruin performance than anything else, so unless you know exactly where the I/O bottleneck is, do big writes into big files, even buffering internally in your code if necessary

- and big writes really means big: a 10 MB write is not a big write, let alone a big file!
- unfortunately, python is not very good at demonstrating this but you can try to compile and run this (available in `codes/cpp/chunk_size_effect.c`)

```
// This file is generated by org-mode, please do not edit
#define _GNU_SOURCE 1
#define _POSIX_C_SOURCE 200809L
#define _XOPEN_SOURCE 700
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define SIZE 1000*1000*100

int main(int argc, char *argv[]) {
  char *file1, *file2;
  if (argc != 3) {
    // please note this is UNSAFE: if such files exist, they will be overwritten
    file1 = "testfile1";
    file2 = "testfile2";
  } else {
    file1 = argv[1];
    file2 = argv[2];
  }
  int fd1 = open(file1, O_WRONLY|O_TRUNC|O_CREAT, S_IRUSR|S_IWUSR);
  int fd2 = open(file2, O_WRONLY|O_TRUNC|O_CREAT, S_IRUSR|S_IWUSR);
  double *data = (double *) calloc(SIZE, sizeof(double));
  struct timespec t1, t2, t3;
  clock_gettime(CLOCK_MONOTONIC, &t1);
  for (int i=0; i<SIZE; i++) {
    write(fd1, data+i, sizeof(double)*1);
  }
  clock_gettime(CLOCK_MONOTONIC, &t2);
  write(fd2, data, sizeof(double)*SIZE);
  clock_gettime(CLOCK_MONOTONIC, &t3);
  printf("Writing one element at a time took %6li seconds\n", t2.tv_sec-t1.tv_sec);
  printf("Writing all elements at once took  %6li seconds\n", t3.tv_sec-t2.tv_sec);
  close(fd1);
  close(fd2);
  return 0;
}
```

```
Writing one element at a time took    102 seconds
Writing all elements at once took       1 seconds
```

- Performant IO is a bit of a dark magic as there are loads of caches on the way from memory to disc and only the limit as file size goes to infinity will measure true IO speed
  - in the above case, my laptop gives 71 and 2 seconds, but 2 s is 4 times the theoretical maximum speed!
- Even more of a dark magic as disc, unlike the CPU, is a shared resource: other users use same discs

## 1.3 Parallel I/O

- always use parallel I/O for parallel programs
- poor man's parallel I/O
  - every worker writes its own file
  - can be the fastest solution
  - but how do you use those files with different number of workers for e.g. post-processing?
- MPI I/O or MPI-enabled HDF5 library deal with that
  - they can write a single file simultaneously from all workers
  - may do some hardware-based optimisations behind the scenes
  - can also map the writes to the MPI topology
  - needs a bit of a learning curve, unless you chose to use h5py or some other library like it which handles the complexity for you

### 1.3.1 Parallel IO with PETSc

```
import sys
import time
import numpy
import mpi4py
from mpi4py import MPI
import petsc4py
petsc4py.init(sys.argv)
from petsc4py import PETSc
import tempfile

dm = PETSc.DMDA().create(dim=3, sizes = (-11,-7,-5),
                         proc_sizes=(PETSc.DECIDE,)*3,
                         boundary_type=(PETSc.DMDA.BoundaryType.GHOSTED,)*3,
                         stencil_type=PETSc.DMDA.StencilType.BOX,
                         stencil_width = 1, dof = 1, comm =
                         PETSc.COMM_WORLD, setup = False)
dm.setFromOptions()
dm.setUp()
vec1 = dm.createGlobalVector()
vec1.setName("NameOfMyHDF5Dataset")
```

```
vec2 = vec1.duplicate()
vec2.setName("NameOfMyHDF5Dataset")
fn = tempfile.NamedTemporaryFile()
vwr=PETSc.Viewer().createHDF5(fn.name, mode=PETSc.Viewer.Mode.WRITE)
vec1.view(vwr)
vwr.destroy()
vwr=PETSc.Viewer().createHDF5(fn.name, mode=PETSc.Viewer.Mode.READ)
vec2.load(vwr)
print("Are they equal? " + ["No!", "Yes!"][vec1.equal(vec2)])
```

```
Are they equal? Yes!
```

- if you ran this in parallel using parallel HDF5 library, you just got all the hard bits for free

### 1.3.2 Parallel IO with h5py

- note that running this in the frontend uses just one rank

```
import mpi4py
from mpi4py import MPI
import h5py
import tempfile
import os
import array
if (MPI.COMM_WORLD.rank == 0):
    temp="files/hdf5_visualisation_example.h5"
else:
    temp=""
KEEP_ME_AROUND = MPI.COMM_WORLD.bcast(temp, root=0)
rank = MPI.COMM_WORLD.rank
f = h5py.File(KEEP_ME_AROUND, "w", driver="mpio", comm=MPI.COMM_WORLD)
dset = f.create_dataset("test", (4,), dtype="f8")
dset[rank] = rank
f.close()
```

- running it from the shell with `mpirun` will use more ranks

```
%%bash
mpirun -np 4 python codes/python/parallel_io_h5py.py
```

- performance might still be bad, because

## 1.4 Know your filesystem

- typical HPDA/HPC system will have a high bandwith, high latency parallel file system where big files should go
- most common is Lustre
  - one often needs to set up a special directory on Lustre for very high bandwidth operations

    - files are *striped* onto different pieces of hardware (OSTs) to increase bandwidth
    - can be tricky as both the number of active OSTs and number of writers in code affect the bandwidth

- in our example, we did not use a distributed file system, so parallellism gave no benefit

    - sorry about that, we would have needed to arrange supercomputer access to demonstrate this: will do on a later course

## 1.5 Checkpointing

- Your code should be able to do this on its own to support solving the problem by running the code several times: often not possible to obtain access to a computer for long enough to solve in one go.

- Basically, you save your iterate or current best estimate solution and later load it from file instead of using random or hard coded initial conditions.

## 1.6 Exercises

### 1.6.1 Experiment with different way so saving a 100x100x100 numpy array

Unfortunately cannot speed-test these easily, but try at least

1. On your own

2. numpy functions

3. h5py

### 1.6.2 Memmapped IO

- Sometimes your file is too big to load into memory, memmap is then your friend.

- Files which have been memmapped, are only loaded into memory a small chunk at a time as it is needed

- But they look like normal files to whoever is using them

- Use h5py's memmap mode and numpy's memmap mode to process (does not matter what you do with it, perhaps just add one) the file you saved above

    - nothing in your code would change if you needed to process the largest file in the world