Parallel Computing in Python: multiprocessing

Konrad HINSEN

Centre de Biophysique Moléculaire (Orléans)

and

Synchrotron Soleil (St Aubin)

Parallel computing: Theory

Parallel computers

- **Multiprocessor/multicore:**
  several processors work on data stored in shared memory
- **Cluster:**
  several processor/memory units work together by exchanging data over a network
- **Co-processor:**
  a general-purpose processor delegates specific tasks to a special-purpose processor (GPU, FPGA, ...)
- **Other:**
  - Cluster of multicore nodes with GPUs
  - NUMA (non-uniform memory access) architectures
  - ...

Almost all computers made today are parallel!

Parallelism vs. concurrency

Parallelism:

use multiple processors to make a computation faster.

Concurrency:

permit multiple tasks to proceed without waiting for each other.

Different goals that share *implementation aspects*.

Scientific computing cares more about ***parallelism***.
Concurrency is rarely needed.

Parallel Programming

- Decomposition of the complete task into independent subtasks and the data flow between them.
- Distribution of the subtasks over the processors minimizing the total execution time.
- For clusters: distribution of the data over the nodes minimizing the communication time.
- For multiprocessors: optimization of the memory access patterns minimizing waiting times.
- Synchronization of the individual processes.

Difficulties

Correctness:

- Verifying that subtasks are indeed independent.
- Making synchronization patterns deadlock-free.
- Clusters: verifying communication patterns.

Efficiency:

- Attributing equal workloads to all processors.
- Taking into account computation *and* communication.
- Optimizing for a specific parallel computer.

Synchronization issues

Deadlock:

- Two processes are waiting for each other to finish.
- Usually caused by **locks** or by **blocking communication**.

Race condition:

- Two or more processes modify a shared resource (variable, file, ...)
- Result depends on which process comes first.
- Can be avoided using **locks**, but...
- ... handling locks is ***very*** difficult and mistakes often cause **deadlocks**.

Task farming

One **master process** supervises the execution of the program. It defines independent **tasks** and puts them on a **to-do list**. It also **collects** the **results** of these tasks.

Any number of **slave processes** each take a task from the to-do list, execute it, and put the result into the master's mailbox.

Advantages:

- Very simple model

- No deadlocks, since only the master process ever waits for another process to finish

Limitations:

- Tasks cannot delegate work to sub-tasks. Adding this possibility would introduce the possibility of deadlocks.

- Rigid communication pattern, no optimization possible.

- Distributed data storage impossible.

Parallel computing: Python practice

Processes vs. threads

A *process* consists of

- a block of memory
- some executable code
- one or more threads that execute code independently but work on the same memory

Multithreading: using *multiple threads* in the *same* process

module threading for concurrency or parallelism

Multiprocessing: using *multiple processes* with *separate* memory

module multiprocessing spaces for concurrency or parallelism

But... why use multiple processes rather than multiple threads?

The Global Interpreter Lock (GIL)

The Python interpreter is not thread-safe.
A few critical internal data structures may only be accessed by one thread at a time. Access to them is protected by the GIL. ***Parallel execution of Python code in different threads is thus impossible.***

This is not a requirement of the Python language, but an implementation detail of the CPython interpreter. Jython and IronPython don't have a GIL and are fully thread-safe.

Attempts at removing the GIL from CPython have failed until now. The main difficulty is maintaining the C API for extension modules and the efficiency for single-thread execution.

Multiprocessing avoids the GIL by having separate processes which each have an independent copy of the interpreter data structures.
The price to pay: serialization of tasks, arguments, and results.

This adds overhead that can be important.

Moreover, ***not all Python objects can be serialized.***

A first parallel program

```
from multiprocessing import Pool
import numpy

if __name__ == '__main__':
    pool = Pool()
    roots = pool.map(numpy.sqrt, range(100))
    print roots
```

for Windows compatibility

What happens...

1. `pool = Pool()` launches one slave process per physical processor on the computer. On Unix systems, the slaves are forked from the master process. Under Windows, a new process is started that imports the script.
2. `pool.map(numpy.sqrt, range(100))` divides the input list into chunks of roughly equal size and puts the tasks (function + chunk) on a todo list.
3. Each slave process takes a task (function + a chunk of data) from the todo list, runs `map(function, chunk)`, and puts the result on a result list.
3. `pool.map` on the master process waits until all tasks are handled and returns the concatenation of the result lists.

The todo list

The todo list is actually a *queue*, i.e. a data structure to which items are added at one end and taken off at the other end.

The todo list must be accessible by all processes (master and slaves).

Access to the todo list must be synchronized to prevent data corruption.

The todo list is stored on the master process. A special thread of the master process waits for task requests from slave processes and returns the task function and arguments. This requires *serialization*.

Explicit task definition

```
from multiprocessing import Pool
import numpy

if __name__ == '__main__':

    pool = Pool()
    results = [pool.apply_async(numpy.sqrt, (x,))
                    for x in range(100)]
    roots = [r.get() for r in results]
    print roots
```

1. pool.apply_async returns a proxy object immediately
2. proxy.get() waits for task completion and returns the result

Use for:

- launching *different* tasks in parallel
- launching tasks with more than one argument
- better control of task distribution

Shared memory

It is possible to share blocks of memory between processes. This eliminates the serialization overhead.

Multiprocessing can create shared memory blocks containing C variables and C arrays. A NumPy extension adds shared NumPy arrays. It it *not* possible to share arbitrary Python objects.

NumPy extension: http://bitbucket.org/cleemesser/numpy-sharedmem

Caveat:

If you care about your mental sanity, **don't modify shared memory contents in the slave processes**. You will end up debugging race conditions.

Use shared memory only to transfer data from the master to the slaves!

Shared memory

```
from multiprocessing import Pool
from parutils import distribute
import numpy
import sharedmem

def apply_sqrt(a, imin, imax):
    return numpy.sqrt(a[imin:imax])

if __name__ == '__main__':

    pool = Pool()
    data = sharedmem.empty((100,), numpy.float)
    data[:] = numpy.arange(len(data))
    slices = distribute(len(data))
    results = [pool.apply_async(apply_sqrt, (data, imin, imax))
               for (imin, imax) in slices]
    for r, (imin, imax) in zip(results, slices):
        data[imin:imax] = r.get()
    print data
```

PARUTILS.DISTRIBUTE

Distributes a sequence equally (as much as possible) over the available processors. Returns a list of index pairs (imin, imax) that delimit the slice to give to one task.

```
from multiprocessing import cpu_count

default_nprocs = cpu_count()
```

```
def distribute(nitems, nprocs=None):
    if nprocs is None:
        nprocs = default_nprocs
    nitems_per_proc = (nitems+nprocs-1)/nprocs
    return [(i, min(nitems, i+nitems_per_proc))
            for i in range(0, nitems, nitems_per_proc)]
```

Shared memory with in-place modification

```
from multiprocessing import Pool
from parutils import distribute
import numpy
import sharedmem

def apply_sqrt(a, imin, imax):
    a[imin:imax] = numpy.sqrt(a[imin:imax])

if __name__ == '__main__':

    pool = Pool()
    data = sharedmem.empty((100,), numpy.float)
    data[:] = numpy.arange(len(data))
    tasks = [pool.apply_async(apply_sqrt, (data, imin, imax))
             for (imin, imax) in distribute(len(data))]
    for t in tasks:
        t.wait()
    print data
```

Debugging: monoprocessing

Parallel debugging is a mess. You don't want to do it.

The module `monoprocessing` contains a class `Pool` with the same methods as `multiprocessing.Pool`, but all tasks are executed immediately and in the same process. This permits debugging with standard tools.

If your programs works with `monoprocessing but not with multiprocessing`, explore the following possibilities:

- Serialization: some object cannot be serialized
- The code of a task refers to a global variable in the master process
- The code of a tasks modifies data in shared memory

Get monoprocessing from: http://pypi.python.org/pypi/monoprocessing/0.1

Exercises

Volume des cellules d'un maillage

Parallélisez le script suivant:

```
import pyShift.cartTh as CTH
import pyShift.volume as PSV
import numpy as np
import time

n = 10
m = CTH.cartThNumpy(n, n, n)

t0 = time.time()
v = PSV.volume(m)
t1 = time.time()

print t1-t0
```

1) Commencez simple (deux processeurs, ...)
2) Mesurez le temps utilisé pour plusieurs n