Parallel Programming with Threads

Overview

This guide will attempt to introduce parallel programming concepts with little assumption of prior experience. We will not be writing code directly, but we will go through a series of examples to illustrate the process. We will be using the simple non-interacting gas simulator as the background project through this and the next guide. All examples will be in Python 3. The `gas.py` simulator uses language features that require Python >= 3.7.

Most of the steps in this guide can be followed through individually as non-root users.

Step 1: Setting up the Python environment & introducing `gas.py`

We will be using the `gas.py` non-interacting gas simulator as an example through this and the next guide. We will need a more modern version of python (>=3.7). We will use `pyenv` to set this up. First, we need to install `pyenv` itself. As your user run:

```
[lowell@te-master ~]$ curl -L
https://raw.githubusercontent.com/yyuu/pyenv-installer/master/bin/pyenv-
installer | bash
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
100  2446  100  2446    0     0   5418      0 --:--:-- --:--:-- --:--:--
5423
Cloning into '/home/lowell/.pyenv'...
...

WARNING: seems you still have not added 'pyenv' to the load path.

# Load pyenv automatically by adding
# the following to ~/.bashrc:
```

```
export PATH="/home/lowell/.pyenv/bin:$PATH"
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"
```

Open your `$HOME/.bash_profile` and add towards the bottom (as suggested by the installer):

```
export PATH="/home/lowell/.pyenv/bin:$PATH"
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"

export PYENV_VIRTUALENV_DISABLE_PROMPT=1
```

We can activate this change right away by "sourcing" our profile:

```
[lowell@te-master ~]$ source .bash_profile
```

We should now have the `pyenv` command available.

```
[lowell@te-master ~]$ pyenv help
Usage: pyenv <command> [<args>]

Some useful pyenv commands are:
   commands    List all available pyenv commands
   local       Set or show the local application-specific Python version
   global      Set or show the global Python version
   shell       Set or show the shell-specific Python version
   install     Install a Python version using python-build
   uninstall   Uninstall a specific Python version
   rehash      Rehash pyenv shims (run this after installing executables)
   version     Show the current Python version and its origin
   versions    List all Python versions available to pyenv
   which       Display the full path to an executable
   whence      List all Python versions that contain the given executable

See `pyenv help <command>' for information on a specific command.
For the full documentation, see: https://github.com/pyenv/pyenv#readme
```

Pyenv allows us to build and maintain multiple version of Python and switch between them. It can also set new global default python installs, as well as set that we should always use a particular python version in a particular directory. We can see what versions it can install with:

```
[lowell@te-master ~]$ pyenv install -l
Available versions:
  2.1.3
```

```
  2.2.3
  2.3.7
  2.4.0
  2.4.1
...
```

The one we will use is `anaconda3-2019.03`. Anaconda is a bundled version of python that comes with various scientific software available. Let's install it (this will take a few minutes):

```
[lowell@te-master ~]$ pyenv install anaconda3-2019.03
Downloading Anaconda3-2019.03-Linux-x86_64.sh...
-> https://repo.continuum.io/archive/Anaconda3-2019.03-Linux-x86_64.sh
Installing Anaconda3-2019.03-Linux-x86_64...
Installed Anaconda3-2019.03-Linux-x86_64 to
/home/lowell/.pyenv/versions/anaconda3-2019.03
```

We can check what's available with:

```
[lowell@te-master ~]$ pyenv versions
* system (set by /home/lowell/.pyenv/version)
  anaconda3-2019.03
```

Let's globally select `anaconda3-2019.03` as our version:

```
[lowell@te-master ~]$ python --version
Python 2.7.5
[lowell@te-master ~]$ pyenv global anaconda3-2019.03
[lowell@te-master ~]$ python --version
Python 3.7.3
```

If you ever want to switch back to the system provided version, use:

```
[lowell@te-master ~]$ pyenv global system
[lowell@te-master ~]$ python --version
Python 2.7.5
```

With `anaconda3-2019.03` enabled, let's move ahead with testing our `gas.py` simulator.

You can find `gas.py` in the bundle `/data/gas.tar.gz` on the CM. Extract this in your home directory, then go into the `gas` directory.

```
[lowell@te-master ~]$ cd gas
[lowell@te-master gas]$ ls
```

```
Box.py  Cell.py  gas.py  Particle.py  split.py  World.py
```

The only file we will be dealing with is `gas.py`. This is where the main logic of the simulator is located and is the script we will execute. Briefly, the other files do the following:

- `Box.py` contains some basic dataclasses that `gas.py` uses
- `Cell.py` contains the logic surrounding Cells and Cell boundary conditions
- `Particle.py` contains particle logic, including most of the physics
- `split.py` is a utility we will make use of later that splits `gas.py` output into multiple files for easier use with ParaView (visualization tool)
- `World.py` contains a class that defines the global characteristics of our simulation

We can see what options `gas.py` provides:

```
[lowell@te-master gas]$ ./gas.py -h
Usage: ./gas.py
   -h, --help       : Print this usage info
   -p, --part <#>   : Number of particles (default: 10)
   -s, --steps <#>  : Number of timesteps (default: 1)
   -o, --out <file> : Output to file (default: stdout)
```

We can specify the number of particles (`-p`), a number of timesteps (`-s`), and an output file for generated data (`-o`).

Gas works like most physics simulators. It divides the simulation into "timesteps". At each timestep, it moves the particles for a small period of time. It then computes if any exceptions like collisions have occurred, and deals with them. Finally, it figures out if any particles have crossed boundaries, and computes their boundary conditions.

For simplicity sake, you may notice that there is no way to feed initial conditions into Gas. Gas always generates a random particle distribution every time it runs.

Most particle simulators use a similar approach. In general, particle simulators would use force interactions. In this case, at each timestep, we first compute all of the forces and derive accelerations from them, then we "integrate" the timestep. The same basic concepts are used.

We can make sure Gas is working by running once with the default options:

```
[lowell@te-master gas]$ ./gas.py
0.0009 running with 10 particles for 1 steps, output to <stdout>
--- ts=0
0.01393473347246452,0.0043420756285396335,2.8700354969971222,8.65078646254
8004,8.902126505175913,0.024024720114272098,0.022034620418221525,0.0171140
9688452525
...
0.0014 timestep=1, particles=10
  0.0014 comb particles into cells
```

```
    Cell 0: 0 particles added to our cell
  0.0015 start integrate
  0.0015 start collide
    Cell 0: resolved 0 collisions
  0.0019 start boundary check
    Cell 0: 0 particles out-of-bounds
  0.0020 resolve out-of-bounds particles
  reflected 0 particles of 0 in the oob list
  0.0020 start print
--- ts=1
0.01393473347246452,0.0043420756285396335,2.871236733002836,8.651888193568
915,8.90298221002014,0.024024720114272098,0.022034620418221525,0.017114096
88452525
...
0.0022 done
```

This is a ***very*** boring simulation. It only computed one timestep, and nothing really happened. But, it worked.

Let's get a little familiarity with the structure of the code. Inside `gas.py`, we find a class called `Sym`. This is where everything we will care about happens. The `run()` method of this class is the main simulation loop. It computes the following steps:`

1. "comb" the set of particles to see what belongs in our Cell (box). This step is a little superfluous now but will prove handy later.
2. "integrate" the timestep, i.e. move particles for a brief instant
3. figure out which particles "collide"; resolve those collisions (i.e. figure out how they scattered off of each other. This bit is handled by the class in `Particle.py`)
4. check to see if any particles went outside of the cell boundaries, make a list of them
5. apply boundary conditions to the particles that left the cell. By default, our boundary conditions are "hard", i.e. the particles bounce off the boundary.

Take a moment to look through this class to see how it works. We'll be making changes to this as we go along.

```
# Simulation loop

class Sym:
    def boundary_resolve(self):
        c = 0
        for p in self.w.oob:
            p.boundary_resolve_hard(self.w.box)
            c += 1
        print("\treflected {} particles of {} in the oob list".format(c,
len(self.w.oob)), file=sys.stderr)

    def run(self):
        c = Cell(0, self.w, parts, self.w.box)
        print("--- ts=0", file=out)
        c.print(out)
        for i in range(1,steps+1):
            print("{:0.4f} timestep={}, particles=
```

```
{}".format(since(self.w), i, len(c.parts)), file=sys.stderr)
            print("\t{:0.4f} comb particles into
cells".format(since(self.w)), file=sys.stderr)
            c.comb_particles()
            print("\t{:0.4f} start integrate".format(since(self.w)),
file=sys.stderr)
            c.integrate()
            print("\t{:0.4f} start collide".format(since(self.w)),
file=sys.stderr)
            c.collide()
            print("\t{:0.4f} start boundary check".format(since(self.w)),
file=sys.stderr)
            c.boundary_check()
            print("\t{:0.4f} resolve out-of-bounds
particles".format(since(self.w)), file=sys.stderr)
            self.boundary_resolve()
            print("\t{:0.4f} start print".format(since(self.w)),
file=sys.stderr)
            print("--- ts={}".format(i), file=out)
            c.print(out)

    def __init__(self, w: World):
        self.w = w
```

Step 2: Visualizing our simulation

We're going to go ahead and run a simple example simulation and visualize it. This will help us check ourselves as we work along. We won't repeat these steps beyond this point, but you can always go back and check with any new simulation we run.

We are going to use ParaView for visualization. Go ahead and download it and install it on your laptop now.

Let's run a simulation. We will keep it small so it runs quickly. We will run with 100 particles for 1000 timesteps. We will output to `sim.csv`. We will redirect stderr to `sim.log` to keep the flood of messages down:

```
[lowell@te-master gas]$ ./gas.py -p 100 -s 1000 -o sim.csv 2> sim.log
```

This should complete in around 30 seconds.

If you look at the `sim.csv` file it's full of lines of comma separated numbers and the occasional `--- ts=#` line. ParaView, that we are going to be using for visualization will work better if this file is split into one file per timestep instead. This is what the `split.py` script is for.

```
[lowell@te-master gas]$ ./split.py
usage: ./split.py <file> <prefix>
```

Let's make a directory called `sim` for these files to go into:

```
[lowell@te-master sim]$ ../split.py ../sim.csv sim
created 1001 timestep files
[lowell@te-master sim]$ ls
sim.1000.csv  sim.1.csv    sim.300.csv
...(LOTS of files)
```

This is a format that ParaView can read in directly. Create a tarball of this directory and move it to your laptop:

```
[lowell@te-master gas]$ tar zcvf sim.tar.gz sim
sim/
sim/sim.1.csv
sim/sim.2.csv
sim/sim.3.csv
sim/sim.4.csv
sim/sim.5.csv
...
sim/sim.1001.csv
```

For the rest of this section we will work through setting up the ParaView visualization interactively. Let the instructor know when you've gotten here.

Step 3: Simple threading examples

Single-threaded

In this section, we'll play with a couple of simple code examples illustrating threading concepts. Make sure you create and run these scripts as we go along, and study their structure.

Let's start with a simple, non-threaded script. This script adds all of the numbers between 1 and 100 (sound familiar?).

```
#!/usr/bin/env python3
# sum.py

sum=0

for i in range(1,101):
  sum += i
  print("{} : {}".format(i, sum))
```

Running this should produce:

```
[lowell@te-master threading-examples]$ ./sum.py
1 : 1
2 : 3
3 : 6
```

```
4 : 10
...
100 : 5050
```

Two threads

We can divide this problem in half with threads. We'll have one thread add the numbers between 1 and 50, and another the numbers between 51 and 100, then add them together in the end.

The general structure for a threaded program is:

1. create the thread and bind it to a function
2. start the threads
3. "join" the threads (i.e. wait for them to stop)
4. finish up

Here's an example that does this:

```
#!/usr/bin/env python3
# sum-2x.py

import threading

low=0
high=0

def sum_low():
  global low
  for i in range(1,51):
    low += i
    print("{} : {}".format(i, low))

def sum_high():
  global high
  for i in range(51,101):
    high += i
    print("{} : {}".format(i, low))

t_low = threading.Thread(target=sum_low)
t_high = threading.Thread(target=sum_high)

t_low.start()
t_high.start()

t_low.join()
t_high.join()

print("total : {}".format(low + high))
```

Running:

```
[lowell@te-master threading-examples]$ ./sum-2x.py
1 : 1
2 : 3
3 : 6
4 : 10
...
total : 5050
```

One interesting thing already is that `sum-2x.py` performs slightly worse than `sum.py`:

```
[lowell@te-master threading-examples]$ time ./sum.py >/dev/null

real    0m0.219s
user    0m0.098s
sys    0m0.133s
[lowell@te-master threading-examples]$ time ./sum-2x.py >/dev/null

real    0m0.235s
user    0m0.113s
sys    0m0.135s
```

This happens because initiating a thread costs more than the win we get from parallelism.

n-threads

Despite the performance loss, let's take this a step further. Let's make a version that takes an argument that specifies how many threads to run.

```
#!/usr/bin/env python3
# sum-n.py

import sys
import threading

s = list()
def sum(id, threads):
  global s
  r = int(100/threads)
  for i in range(r * id + 1 ,r * (id+1) + 1):
    s[id] += i
    print("{} : {}".format(i, s[id]))

# entry
if len(sys.argv) != 2:
  print("Usage: {} <num_threads>".format(sys.argv[0]))
  sys.exit()

threads = int(sys.argv[1])
```

```
print("Running with {} threads".format(threads))

t = list()
for i in range(0,threads):
  t.append(threading.Thread(target=sum, args=(i, threads,)))
  s.append(0)

for i in range(0,threads):
  t[i].start()

total = 0
for i in range(0,threads):
  t[i].join()
  total += s[i]

print("total : {}".format(total))
```

This one is a bit more complicated, but it adds a couple of core ideas:

1. the model of making a list of threads and starting/joining them is a common one
2. we can pass arguments to our thread as well
3. we avoided colliding in our sums by storing every thread's results in a unique element of a list

Let's run this code, say, with 2 threads:

```
[lowell@te-master threading-examples]$ ./sum-n.py 2
Running with 2 threads
1 : 1
2 : 3
3 : 6
...
total : 5050
```

Looks good. Let's try 6 threads:

```
[lowell@te-master threading-examples]$ ./sum-n.py 6
Running with 6 threads
1 : 1
2 : 3
3 : 6
...
96 : 1416
total : 4656
```

Wait, we got a different answer. Why? Think of how we constructed our ranges.

Let's do a quick scaling study:

```
[lowell@te-master threading-examples]$ for i in $(seq 1 20); do time
./sum-n.py $i >/dev/null; done 2>&1 | grep real
real    0m0.233s
real    0m0.242s
real    0m0.246s
real    0m0.243s
real    0m0.225s
real    0m0.225s
real    0m0.229s
real    0m0.230s
real    0m0.236s
real    0m0.229s
real    0m0.236s
real    0m0.239s
real    0m0.231s
real    0m0.226s
real    0m0.228s
real    0m0.206s
real    0m0.224s
real    0m0.228s
real    0m0.231s
real    0m0.233s
```

The threading isn't doing us any favors in this example. In general, threading only is an advantage if the time it takes to make the threads and load the libraries is much smaller than the time each thread spends running. Also, as a side note, Python threading just isn't very good.

Locking

In the example above, we kept track of the sums by putting them all in an item-per-thread array (`s`) and summing them at the end. If we instead using ***locking***, we can actually let the threads to all of the summing and have only one global sum.

```
#!/usr/bin/env python3
# sum-n-lock.py

import sys
import threading


lock = threading.Lock()
total = 0

def sum(id, threads):
  global total
  r = int(100/threads)
  sum = 0
  for i in range(r * id + 1 ,r * (id+1) + 1):
    sum += i
    print("{} : {}".format(i, sum))
```

```
    lock.acquire()
    total += sum
    lock.release()

  # entry
  if len(sys.argv) != 2:
    print("Usage: {} <num_threads>".format(sys.argv[0]))
    sys.exit()

  threads = int(sys.argv[1])
  print("Running with {} threads".format(threads))

  t = list()
  for i in range(0,threads):
    t.append(threading.Thread(target=sum, args=(i, threads,)))

  for i in range(0,threads):
    t[i].start()

  for i in range(0,threads):
    t[i].join()

  print("total : {}".format(total))
```

This is an example of a mutex lock strategy, though Python calls it acquire/release rather than lock/unlock. Specifically, look at these lines:

```
    lock.acquire()
    total += sum
    lock.release(
```

We know that the `total` variable is safe because it only gets touched when the lock is held, so only one thread at a time is allowed to write to it.

Step 4: Failure modes for parallel codes

Let's look at two quick examples that illustrate race conditions and deadlocks.

Race condition

We're going to intentionally modify `sum-2x.py` to have a race condition. It's not that hard to do. All we have to do is wait for only ***one*** of the threads to join:

```
  #!/usr/bin/env python3
  # sum-race.py

  import threading

  low=0
```

```
high=0

def sum_low():
  global low
  for i in range(1,51):
    low += i
    print("{} : {}".format(i, low))

def sum_high():
  global high
  for i in range(51,101):
    high += i
    print("{} : {}".format(i, low))

t_low = threading.Thread(target=sum_low)
t_high = threading.Thread(target=sum_high)

t_low.start()
t_high.start()

t_low.join()

print("total : {}".format(low + high))
```

Let's try running a couple of times. The first time I ran it I got:

```
[lowell@te-master threading-examples]$ ./sum-race.py
1 : 1
2 : 3
3 : 6
...
50 : 1275
total : 5050
```

That looks fine. Then I ran it again:

```
[lowell@te-master threading-examples]$ ./sum-race.py
1 : 1
2 : 3
3 : 6
...(middle of the file)
50 : 1275
total : 1653
57 : 861
...
99 : 1275
100 : 1275
```

That doesn't look good at all. What's happening here?

case 1) By not waiting for the `high` thread to join, by random chance, we might have the situation that `high` completed before `low`. In this case, everything is ***fine*** because `high` has already finished.

case 2) However, we also have the chance that `low` finishes before `high`. In this unfortunate case, we go on to add the total and try to clean up before `high` has finished its work.

In other words, we have a ***race*** between `high` and `low`. It's up to chance whether or not our code runs as expected.

Race conditions can be notoriously difficult to track down. They almost always result from a failure to ***synchronize*** our threads/processes appropriately, but then go on to assume that they were synchronized. In our case, our missed synchronization is with the joins. In the final version of `gas.py` (next guide), we'll see another form called a `barrier`. A `barrier` is a call that all threads must make. All threads will wait until the last thread as called `barrier` and then they will move on.

Deadlock

We can use the `sum-n-lock.py` to illustrate a deadlock:

```
#!/usr/bin/env python3
# sum-n-deadlock.py

import sys
import threading


lock = threading.Lock()
total = 0

def add_to_total(i):
  global total
  lock.acquire()
  total += i
  lock.release()

def sum(id, threads):
  r = int(100/threads)
  sum = 0
  for i in range(r * id + 1 ,r * (id+1) + 1):
    sum += i
    print("{} : {}".format(i, sum))
  lock.acquire()
  add_to_total(sum)
  lock.release()

# entry
if len(sys.argv) != 2:
  print("Usage: {} <num_threads>".format(sys.argv[0]))
  sys.exit()
```

```
threads = int(sys.argv[1])
print("Running with {} threads".format(threads))

t = list()
for i in range(0,threads):
  t.append(threading.Thread(target=sum, args=(i, threads,)))

for i in range(0,threads):
  t[i].start()

for i in range(0,threads):
  t[i].join()

print("total : {}".format(total))
```

We've modified our code to offload the summing of the total to the function `add_to_total`. In principle, this can be a good idea (more on this in a minute). Let's see what happens when we run it:

```
[lowell@te-master threading-examples]$ ./sum-n-deadlock.py 2
Running with 2 threads
1 : 1
2 : 3
...
50 : 1275
^CTraceback (most recent call last):
  File "./sum-n-deadlock.py", line 43, in <module>
    t[i].join()
  File "/home/lowell/.pyenv/versions/anaconda3-
2019.03/lib/python3.7/threading.py", line 1032, in join
    self._wait_for_tstate_lock()
  File "/home/lowell/.pyenv/versions/anaconda3-
2019.03/lib/python3.7/threading.py", line 1048, in _wait_for_tstate_lock
    elif lock.acquire(block, timeout):
KeyboardInterrupt
^CException ignored in: <module 'threading' from
'/home/lowell/.pyenv/versions/anaconda3-
2019.03/lib/python3.7/threading.py'>
Traceback (most recent call last):
  File "/home/lowell/.pyenv/versions/anaconda3-
2019.03/lib/python3.7/threading.py", line 1281, in _shutdown
    t.join()
  File "/home/lowell/.pyenv/versions/anaconda3-
2019.03/lib/python3.7/threading.py", line 1032, in join
    self._wait_for_tstate_lock()
  File "/home/lowell/.pyenv/versions/anaconda3-
2019.03/lib/python3.7/threading.py", line 1048, in _wait_for_tstate_lock
    elif lock.acquire(block, timeout):
KeyboardInterrupt
```

It hung forever, and I had to `^C` to kill it. What went wrong?

Well, we got the lock, then we called `add_to_total`, which in turn asked for the lock... but we already have the lock. So, it deadlocked. It would hang like this forever waiting for the lock to clear because of the circular locking dependency.

If we simply remove the locking statements from the `sum` method and keep them in the `add_to_total` method, this can be a good way to actually avoid deadlocks. The idea is: 1) to move the locking close to the operation that needs to be locked; 2) restrict access to shared variables to functions that do locking for you. This makes the operations behave almost like atomic operations (i.e. cannot be interrupted).

Step 5: Multi-threaded `gas.py`

Now that we've seen some basics of how threads work in python, let's check out the threaded version of Gas. The threaded version is in a different branch. We can just check it out:

```
[lowell@te-master gas]$ git checkout threaded
Switched to branch 'threaded'
```

Let's look at the updated `Sym` object:

```
class Sym(threading.Thread):
    def boundary_resolve(self):
        c = 0
        for p in self.w.oob:
            p.boundary_resolve_hard(self.w.box)
            c += 1
        print("\treflected {} particles of {} in the oob list".format(c,
len(self.w.oob)), file=sys.stderr)

    def run_cell(self, c):
        print("\t{:0.4f} comb particles into cells".format(since(self.w)),
file=sys.stderr)
        c.comb_particles()
        print("\t{:0.4f} start integrate".format(since(self.w)),
file=sys.stderr)
        c.integrate()
        print("\t{:0.4f} start collide".format(since(self.w)),
file=sys.stderr)
        c.collide()
        print("\t{:0.4f} start boundary check".format(since(self.w)),
file=sys.stderr)
        c.boundary_check()

    def print(self):
        for c in self.cells:
            c.print(self.w.out)

    def run(self):
        for i in range(0,threads):
            zrange = (self.w.box.z.high - self.w.box.z.low)/threads
```

```
            c = Cell(i, self.w, int(parts/threads), Box(self.w.box.x,
self.w.box.y, Range(i*zrange, (i+1)*zrange)))
            self.cells.append(c)

        # create a cell
        print("Running with {} threads".format(threads))
        print("{:0.4f} running with {} particles for {} steps, output to
{}".format(since(self.w), parts, steps, out.name), file=sys.stderr)

        print("--- ts=0", file=out)
        self.print()

        for i in range(1, steps+1):
            tds = list()
            for c in self.cells:
                print("{:0.4f} timestep={}, particles=
{}".format(since(self.w), i, len(c.parts)), file=sys.stderr)
                tds.append(threading.Thread(target=self.run_cell, args=
(c,)))

            for td in tds:
                print("starting thread")
                td.start()

            for td in tds:
                td.join()

            print("\t{:0.4f} resolve out-of-bounds
particles".format(since(self.w)), file=sys.stderr)
            self.boundary_resolve()
            print("\t{:0.4f} start print".format(since(self.w)),
file=sys.stderr)
            print("--- ts={}".format(i), file=out)
            self.print()

        print("{:0.4f} done".format(since(self.w)), file=sys.stderr)
        out.close()


    def __init__(self):
        self.cells = list()
        self.w = World(steps=steps, out=out, parts=parts, cells=cells)
```

There are quite a few changes here. This is how it works:

1. for each thread we create a cell
2. each cell gets a slice of the box, where we make `threads` slices along the z-axis
3. threads are managed and launched much like our `sum-n.py`, in arrays
4. individual cells are combed, integrated, collided and boundary detected in their threads. When particles leave a cell they're added to an `oob` (out-of-bounds) list
5. after all threads rejoin, the main process works out boundary resolution for any `oob` particles that are out-of-bounds for the whole simulation and continues to the next loop iteration

Like `sum-n.py`, we can choose an arbitrary number of threads. Note that in this version, the "combing" operation is actually necessary. That's how a cell figures out if any of the `oob` particles entered its region. If they did, "comb" claims them and removes them from the `oob` list.

Note that we now have a `-t` option to specify the number of threads:

```
[lowell@te-master gas]$ ./gas.py -h
Usage: ./gas.py
    -h, --help       : Print this usage info
    -p, --part <#>   : Number of particles (default: 10)
    -s, --steps <#>  : Number of timesteps (default: 1)
    -o, --out <file> : Output to file (default: stdout)
    -t, --threads <file> : Number of threads (default: 1)
```

Let's do a quick scaling study with 1, 2, and 4 threads.

```
[lowell@te-master gas]$ for i in 1 2 4; do echo $i; time ./gas.py -t$i -
p2048 -s 4 >/dev/null 2>&1; done
1

real    1m10.642s
user    1m10.482s
sys    0m0.171s
2

real    1m25.563s
user    1m13.799s
sys    0m46.325s
4

real    1m19.427s
user    0m57.309s
sys    1m13.709s
```

Why are these performing so badly? It's actually taking longer to run more threads!

Let's use the `strace` command to investigate. `strace` is an extremely handy command. Generally, it gives a listing of every syscall a command run underneath it makes. Try this:

```
[lowell@te-master gas]$ strace ls
execve("/usr/bin/ls", ["ls"], [/* 71 vars */]) = 0
brk(NULL)                               = 0x18c3000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f3e0fb79000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
directory)
open("/opt/ohpc/pub/mpi/openmpi3-
gnu8/3.1.3/lib/tls/x86_64/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1
```

```
ENOENT (No such file or directory)

...(lots more lines)

fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f3e0fb78000
write(1, "Box.py\tCell.py  gas.py  Particle"..., 69Box.py    Cell.py
gas.py  Particle.py  __pycache__  split.py  World.py
) = 69
close(1)                                = 0
munmap(0x7f3e0fb78000, 4096)            = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

This can be a very hand way to see why a program is failing, for instance. E.g., did it try to open a file it couldn't open?

`strace` also has a mode that is like a light-weight profiler for syscalls. If you run it with `-c` it will give you statistics on where it spent its time in syscalls:

```
[lowell@te-master gas]$ strace -c ls
Box.py    Cell.py  gas.py  Particle.py  __pycache__  split.py  World.py
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 19.94    0.000254           9        29           mmap
 19.54    0.000249           8        33        22 open
 18.05    0.000230          13        18           mprotect
 ...
```

Let's use this to see what `gas.py` is up to. Let's start with 1 thread:

```
[lowell@te-master gas]$ strace -fc ./gas.py -t1 -p2048 -s 4 -o sym
strace: Process 12878 attached
...
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 59.99    0.083590        1007        83        39 wait4
  7.27    0.010132           5      1872       737 stat
  5.94    0.008277           9       921       471 open
  5.73    0.007990           4      2243           rt_sigprocmask
...
```

Now, for 4 threads:

```
[lowell@te-master gas]$ strace -fc ./gas.py -t4 -p2048 -s 4 -o sym
strace: Process 13006 attached
...
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 99.98  514.903024          40  12806837   3159288 futex
  0.01    0.068111         821        83        39 wait4
```

That's right... it spent 99.98% of execution time in a `futex` syscall. A `futex` is a kind of lock that can be requested at the kernel level, in some ways like a mutex.

What's this all about? We are likely the victim of the Python "Global Interpreter Lock" (GIL). See, for instance: Thread State and the Global Interpreter Lock. This is a Python-specific problem, and our code would likely have scaled fairly well in some other languages.