High Performance Computing Concepts

Demitri Muna

4 August 2017

High Performance Computing

Virtually all programs you write run as a single process on a single CPU. Increases in speed from one generation of CPU to next has significantly slowed, partly as the focus has moved to use less power. However, your computer has:

- multiple cores
- GPU (with dozens to hundreds of cores)
- hardware vector acceleration
- specialized CPUs

In addition, most of you have access to large clusters of computers, each with all of the above.

Taking full advantage of these requires a new set of skills, but also require a deeper understanding of the underlying hardware and software.

Concepts

This lecture is an introduction to some of these concepts, not a recipe on how to fully use them (which could easily be more than a full semester course each).

multiprocessing

vector operations

SSE

forking

multithreading

GIL

OpenCL

Xeon Phi

Intel IPP

MPI

OpenGL

GPU

three iPhones duct taped together

CUDA

What is a Process?

The `top` program lists all of the programs running on your (a) computer.
Some are graphical (e.g. iTunes), some are *faceless background* programs (e.g. a web server).

demitri — top — 80×24

```
Processes: 327 total, 2 running, 16 stuck, 309 sleeping, 1741 threads  00:40:09
Load Avg: 2.21, 2.02, 1.94  CPU usage: 7.53% user, 13.24% sys, 79.22% idle
SharedLibs: 102M resident, 0B data, 12M linkedit.
MemRegions: 315669 total, 1907M resident, 46M private, 407M shared.
PhysMem: 8139M used (1787M wired), 50M unused.
VM: 901G vsize, 1351M framework vsize, 148012783(127) swapins, 150641681(0) swap
Networks: packets: 11541504/11G in, 10180656/1518M out.
Disks: 19151189/762G read, 9351629/752G written.

PID    COMMAND      %CPU TIME     #TH   #WQ  #PORTS MEM    PURG  CMPRS PGRP
36267  mdworker     2.4  00:00.06 6     3    64+    1888K+ 0B    0B    36267
36266  CVMCompiler  0.0  00:00.27 2     1    28     12M    0B    3048K 36266
36264  Grab         0.0  00:00.74 9     6    206-   31M-   6708K 1648K 36264
36263  top          8.2  00:02.40 1/1   0    22     4056K  0B    1348K 36263
36249  bash         0.0  00:00.02 1     0    15     208K   0B    480K  36249
36248  com.apple.Ac 0.0  00:00.01 2     1    21     32K    0B    848K  36248
36247  login        0.0  00:00.12 2     0    27     64K    0B    1072K 36247
36246  QuickLookSat 0.0  00:00.26 3     1    41     2196K  0B    9484K 36246
36245  quicklookd   0.0  00:00.55 4     0    89     5792K  0B    4416K 36245
36224  com.apple.iC 0.0  00:00.48 2     0    48     260K   0B    2320K 36224
36222  com.apple.Ma 0.0  00:00.34 3     0    127-   1296K- 0B    3948K 36222
36205  com.apple.au 0.0  00:00.01 2     1    32     8192B  0B    1000K 36205
36195  mdworker     0.0  00:00.61 3     0    58     1288K  0B    2068K 36195
36186  printtool    0.0  00:00.03 2     1    34     24K    0B    1208K 36186
```

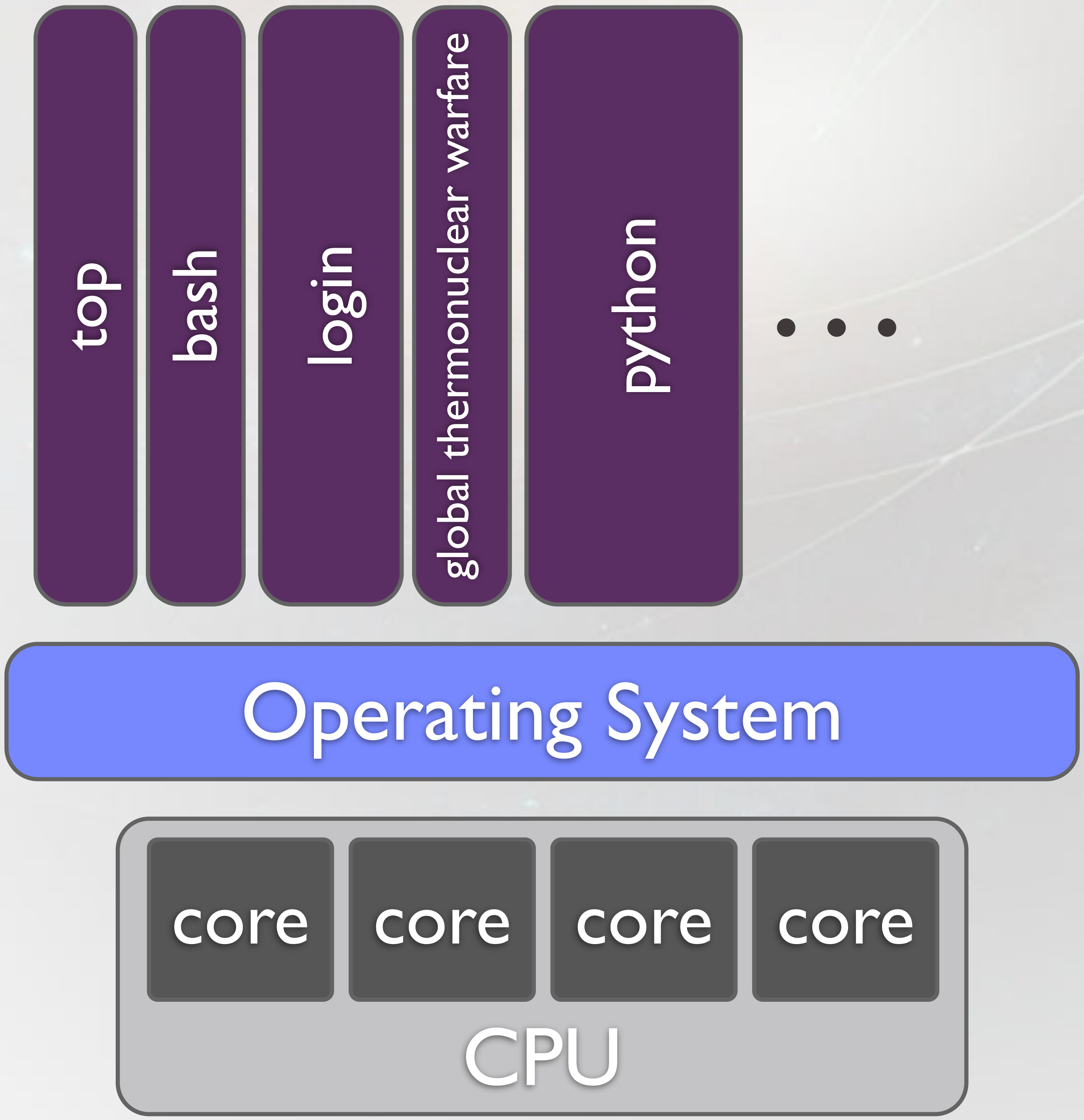
total system memory

on Linux, type "1" in this screen to see how many CPU cores there are

process ID (PID)

number of threads per process

Processes



- The operating system manages each process.
- Each process is independent: memory cannot be shared or accessed between them.
- Each process takes up a different amount of memory.
- Each process has a unique ID (PID).

possibly one of my worst diagrams ever

Threads

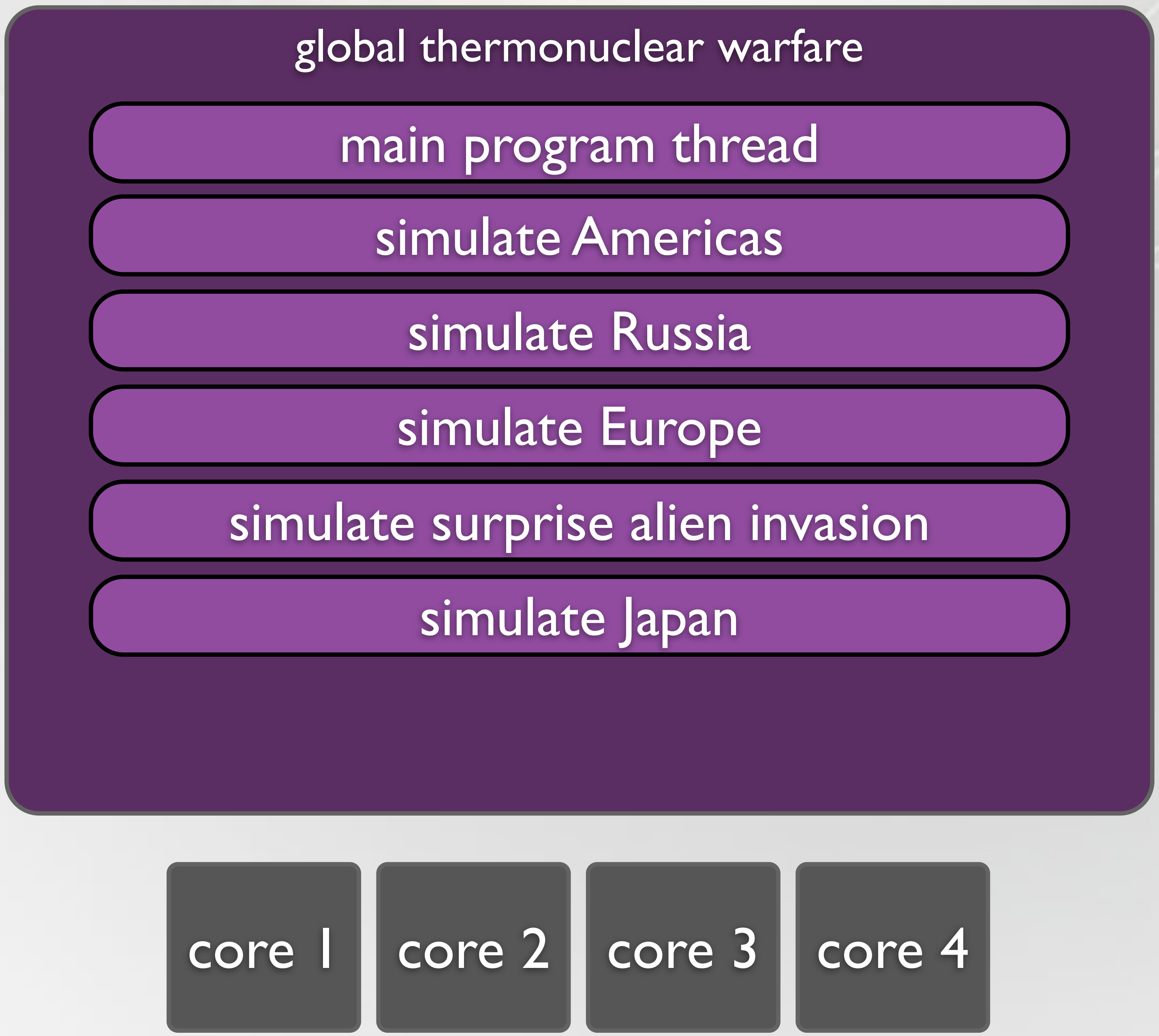
A process can consist of one or more *threads*.

- The process will have a *main thread*, the one created when the program started.
- With one thread, the program can only do one thing at a time. With two threads, the program can perform two tasks (e.g. call two different functions) at the same time.
- The operating system runs each thread on its own core (hence the simultaneous execution).
- Threads in a process have full access to the memory of that process.

Threads

A process can consist of one or more *threads*.

one process



- More threads than cores? The OS will juggle them. When a thread gets a core, it runs. The OS will "pause" a thread to run another one.
- The OS *always* has more threads than cores – each process on the computer has one or more threads, so the OS is juggling dozens or hundreds of threads at a time.
- Most threads are not always active, so the OS doesn't given them time they don't need.

Threads vs Processes

Multiprocessing

- running multiple processes, each *independent*
- can't share memory

Multithreading

- can have many threads in the same process, each with their own CPU core
- same process, thus can share memory

Threads

All code you've (probably) written is single process, single core (e.g. Python scripts). Even if you have 12 cores on your computer, a single-threaded process won't run any faster than if you had a single core.

Why don't we write multi-threaded programs?

It's *hard*.

Threadsafe Code

Most code is not *thread-safe*. Consider this class:

```
class Results(object):
    def __init__(self):
        self.results = np.zeros(100)
        self.counter = 0

    def add_result(self, value):
        # add new value at position
        # kept by counter
        self.results[counter] = value

        # update counter
        self.counter = self.counter + 1

    def number_of_results(self):
        return self.counter
```

It might be called like this:

```
results = Results()
results.add_result(calculation1())
results.add_result(calculation2())
results.add_result(calculation2())
```

Reasonable enough.

Threadsafe Code

Now consider two different threads calling `add_result` at the same time. Remember threads can be paused or run at any time by the system.

```
class Results(object):
    def __init__(self):
        self.results = np.zeros(10)
        self.counter = 0

    def add_result(self, value):
        # add new value at position
        # kept by counter
        self.results[counter] = value

        # update counter
        self.counter = self.counter + 1

    def number_of_results(self):
        return self.counter
```

counter = 0

Thread 1 (value=12): `self.results[0] = 12`

Thread 1 paused by system

Thread 2 (value=42): `self.results[0] = 42`

Thread 2: counter = `counter + 1` (→ 1)

Thread 1 resumes

Thread 1: counter = `counter + 1` (→ 2)

Thread 1 (value=99): `self.results[2] = 99`

Thread 1: counter = `counter + 1` (→ 3)

Expected result: [12, 42, 99, 0, 0, 0, 0, 0, 0, 0]

Actual result: [42, 0, 99, 0, 0, 0, 0, 0, 0, 0]

This function is not threadsafe!

Threadsafe Code

Most code is *not* threadsafe. There are ways to fix that example function so that it is, but things get very complicated very quickly, and become *very* hard to debug. Most expert programmers' advice when asked how to write multithreaded code is: "don't".

Luckily, most astronomer's problems are *embarrassingly parallel*. If you are processing 10,000 galaxies (for example) on an eight core computer, divide the galaxies into eight groups and run eight instances of your programs. No threading issues, easy to debug, and you are using all eight cores. This works because the result of one galaxy is not dependent on the result of another (hence the "parallel").

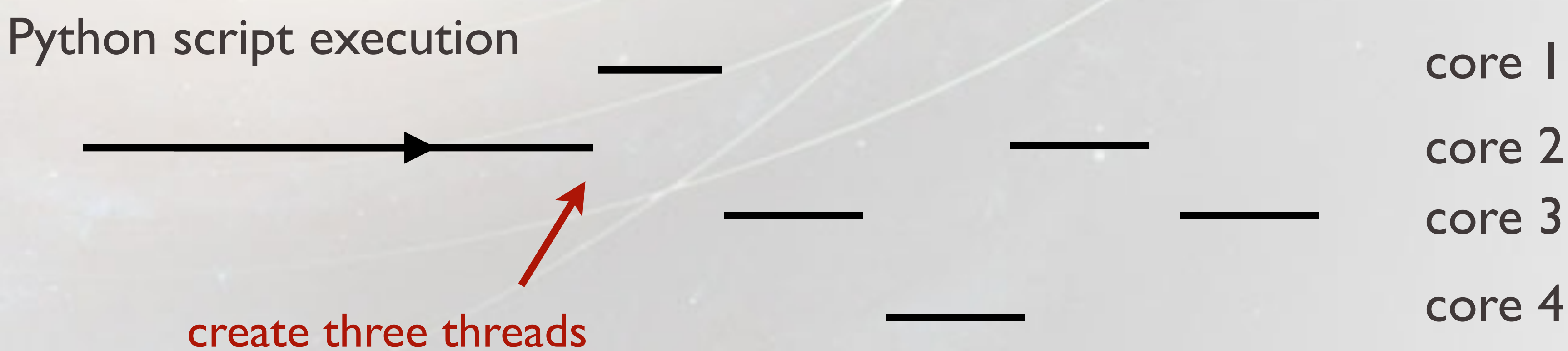
Running in this mode requires the data/processes to be managed though, and this involves extra work. Can we make Python multi-threaded?

Yeeeeee– no.

GIL

There *is* a multithreading package in Python.

There is also GIL: the global interpretive lock.



While technically there are four threads, the GIL prevents more than one from running at any time…

…which kind of defeats the whole purpose.

Thanks, GIL

This is a fundamental feature of Python; it can't be fixed without a low-level reimplementation of Python (i.e. don't hold your breath).

Python Multiprocessing

What about multiprocessing? There is in fact a multiprocessing package.
How does multiprocessing work?

1. Start a Python script (one process).

- your script
- memory you've allocated
- Python interpreter

core 1

2. spawn two processes.

- your script
- memory you've allocated
- Python interpreter

core 2

- your script
- memory you've allocated
- Python interpreter

core 3

Three interpreters, three copies of memory!
But we are using three cores now.

Python Multiprocessing

Creating a new process is called *forking*. It takes the whole process, copies it (memory, code, everything) as a new process, and it gets a new process ID.

PID 4263

- your script
- memory you’ve allocated
- Python interpreter

core 1

PID 4285

- your script
- memory you’ve allocated
- Python interpreter

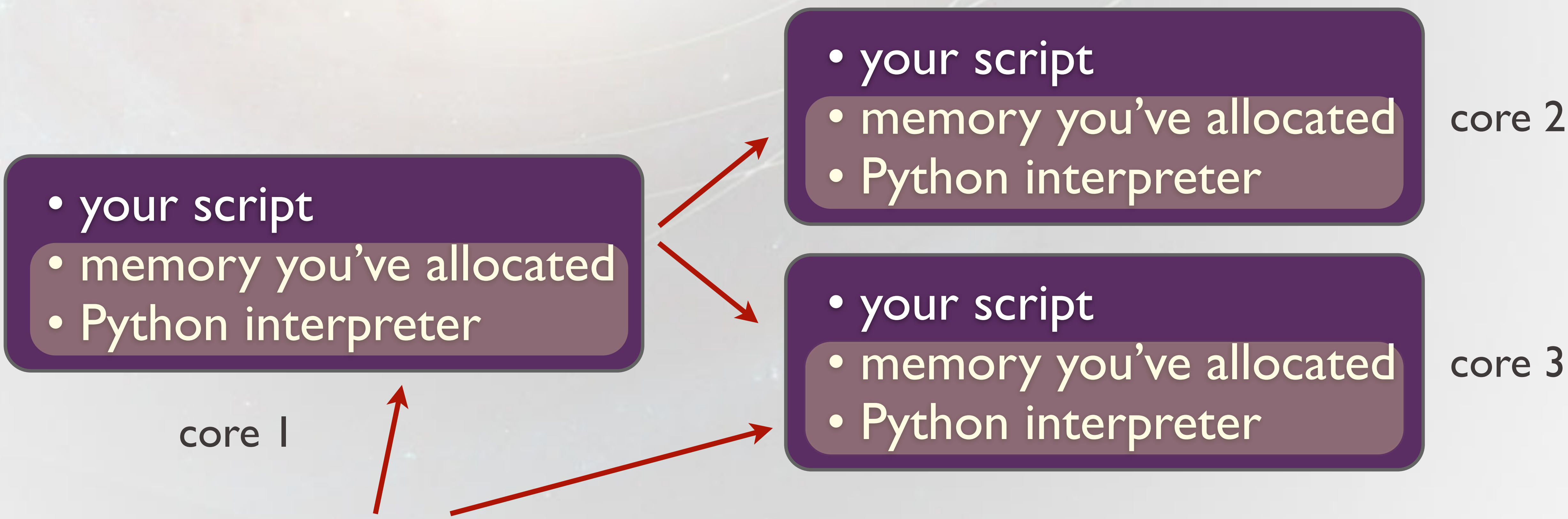
core 2

- your script
- memory you’ve allocated
- Python interpreter

PID 4286

core 3

Python Multiprocessing



This seems wasteful - aren't these the same? But you can't read memory across processes. However, some OS's will recognize this and allow you to read the memory *until* you try to write to it. This feature is called *copy on write*. Here, the Python interpreter (and maybe a large read-only library you read in) are not duplicated.

Vector Programming

Imagine you are adding two arrays. Since a CPU can perform only one operation at a time (per clock cycle), it would look like this:

5	+	5	one clock cycle	10	
3	+	5	one clock cycle	8	
8	+	5	one clock cycle	13	
2	+	5	one clock cycle	7	
7	+	5	one clock cycle	12	time
2	+	5	one clock cycle	7	
7	+	5	one clock cycle	12	
9	+	5	one clock cycle	14	
3	+	5	one clock cycle	8	

9 clock cycles total

(Technically, this is a simplification; it would take more cycles than this, e.g. extra steps to move the values in and out of the CPU.)

Vector Programming

Starting in the 90s, Intel (and others) started creating *vector extensions*. These are operations that can take two vectors of numbers and add them together in *one* clock cycle:

5		5		10
3		5		8
8		5		13
2		5		7
7	+	5	one clock cycle →	12
2		5		7
7		5		12
9		5		14
3		5		8

Examples of vector extensions are MMX, SSE, AVX (Intel) and AltiVec (PPC).

~9x speedup!

Actual speedup varies. 6x is easily attainable, often much higher. The real gains are not in adding a single vector of nine values, but repeating this on a large block of memory (thousands of values).

Vector Programming

- Length of vectors that operate all at once depends on CPU generation. Typical values are 8-16 operations at a time.
- Require use of specific vector functions.
- Vector functions can take large blocks of memory (e.g. an image).
- Not cross platform - functions are CPU dependent (functions tend to be added, not removed).
- Many operations are available, e.g. multiply a vector by a scalar, divide a vector by a scalar, find the min/max values of a vector, average, sum, etc.
- Typically only works on float, double values - *definitely* not objects.
- Modern compilers can identify places in code where this can be used and automatically generate code to do so.
- Very easy to use; mostly just importing a header and calling a specialized function.
- Should check computer you are running on if function is available and fall back if not.

Directory of functions available on Intel platform:

https://software.intel.com/sites/landingpage/IntrinsicsGuide/