Scientific Software Development with Python

High performance computing with Python

Simon Pfreundschuh
Department of Space, Earth and Environment

CHALMERS
UNIVERSITY OF TECHNOLOGY

Agenda

1. Overview

2. Computer architecture

3. Example: Solving the heat equation

4. Runtime profiling

5. Parallel computing

6. Summary

Overview

	Conceptual	Technical
Organisational	Project planning & management	Version control, testing, deployment (DevOps)
Implementational	Software design	Python programming, scientific computing Part 3

Agenda

1. Overview

2. **Computer architecture**

3. Example: Solving the heat equation

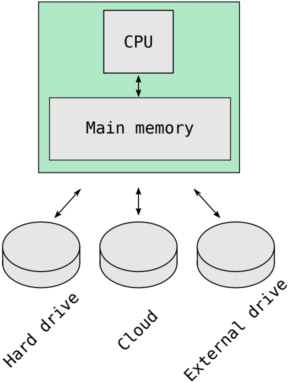
4. Runtime profiling

5. Parallel computing

6. Summary

A (rough) model of a computer system

- Main components: Central processing unit (CPU) and main memory
- External components: Hard drive, external storage, …

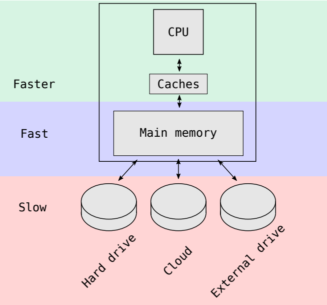


Why does computing take time?

1. Data needs to be loaded
    - Data needs to be transferred from external storage into main memory
2. Data needs to be processed
    - Data is moved from main memory into CPU registers
    - CPU performs calculation
    - Results are stored back into main memory

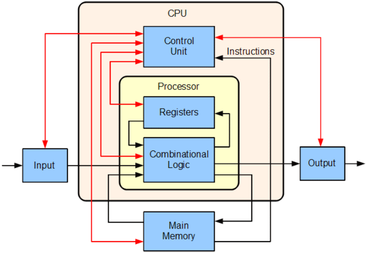
Loading data

- Modern CPUs use a hierarchy of caches to speed up memory transfer between main memory CPU
- Loading data from external drives (even hard disk) is extremely slow.



Processing data

- The CPU (generally) performs all calculations
- Instructions and input data are both read from main memory
- Instructions must given as *machine code* and adhere to the procecessors *instruction set*.
- The number of instructions that can be processed per time is limited by the CPUs clock rate and how many operations it can perform in parallel.
- A lot of complicated things going on to make processing fast: Pipelining, vectorization, branch prediction, . . .

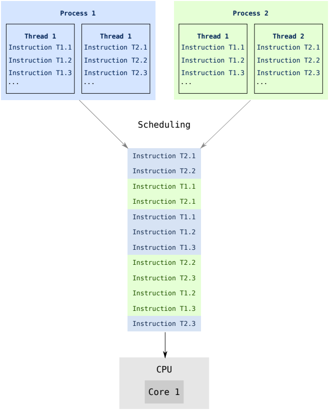


Multitasking and multiprocessing

CHALMERS
UNIVERSITY OF TECHNOLOGY

Multitasking

- Modern operating systems organize program execution into processes and threads that are executed intermittently.
- Each process has its own memory and resources.
- Threads of a common process share memory and resources.
- Multitasking improves performance even on single-core computers because because other threads execute while waiting for data.



Process 1

Thread 1
Instruction T1.1
Instruction T1.2
Instruction T1.3
...

Thread 1
Instruction T2.1
Instruction T2.2
Instruction T2.3
...

Process 2

Thread 1
Instruction T1.1
Instruction T1.2
Instruction T1.3
...

Thread 2
Instruction T2.1
Instruction T2.2
Instruction T2.3
...

Scheduling

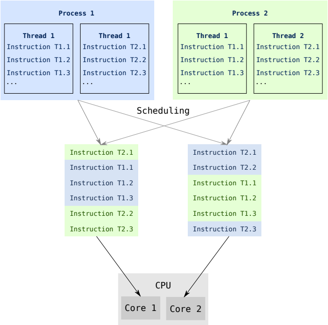
Instruction T2.1
Instruction T2.2
Instruction T1.1
Instruction T2.1
Instruction T1.1
Instruction T1.2
Instruction T1.3
Instruction T2.2
Instruction T2.3
Instruction T1.2
Instruction T1.3
Instruction T2.3

CPU

Core 1

Multiprocessing

- Modern CPUs typically have multiple cores that allow parallel processing of multiple instruction streams.
- This will improve performance for *compute-limited* problems.



Process 1

Thread 1
Instruction T1.1
Instruction T1.2
Instruction T1.3
...

Thread 1
Instruction T2.1
Instruction T2.2
Instruction T2.3
...

Process 2

Thread 1
Instruction T1.1
Instruction T1.2
Instruction T1.3
...

Thread 2
Instruction T2.1
Instruction T2.2
Instruction T2.3
...

Scheduling

Instruction T2.1
Instruction T1.1
Instruction T1.2
Instruction T1.3
Instruction T2.2
Instruction T2.3

Instruction T2.1
Instruction T2.2
Instruction T1.1
Instruction T1.2
Instruction T1.3
Instruction T2.3

CPU
Core 1
Core 2

Some terminology

Sequential: Code that is executed in the order in which it is written. This is (luckily) the standard behavior.

Concurrent: Code that has been designated to be executed in arbitrary order.

Parallelism: Parallelism is achieved when a concurrent program is executed on a multi- core CPU.

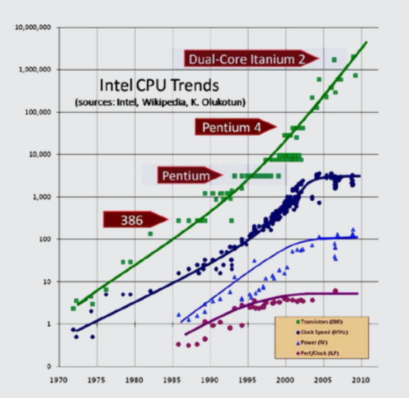
CPU-bound / Compute-bound: Code whose performance is limited by how many calculations can be performed in the CPU

IO-bound: Code that is limited by loading of data from disk or interactions with the OS, both of which are very slow.

Embarassingly parallel: A problem that can be parallelized by simply splitting the input space. Can be parallelized by simply running it with different input in different processes.

Multiprocessing

- Moore’s law: The number of transistors per microprocessor (green) has been increasing exponentially since the 70s.
- But: Clock rate (blue) has been stalling since the 2000s.
- Current computing trends: Massively parallel
  - Need to go parallel to go faster.



Agenda

CHALMERS
UNIVERSITY OF TECHNOLOGY

1. Overview

2. Computer architecture

3. **Example: Solving the heat equation**

4. Runtime profiling

5. Parallel computing

6. Summary

The 2D heat equation

- Let $u(t, x, y)$ describe a time-dependent distribution of heat on a two dimensional euclidean coordinate system.
- Its temporal evolution is described by the *heat equation*:

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y} \right) \tag{1}$$

- Given u at $t = 0$ we can use (1) to compute u at arbitrary time $t = T$.

Numerical solution

1. At each time step t we approximate u using a two dimensional array $U_{i,j}^{t}$ (**domain discretization**):

$$U_{i,j}^{t} = u(t, i \cdot \Delta x, j \cdot \Delta y) \tag{2}$$

2. Replace derivatives with finite differences (**finite difference approximation**):

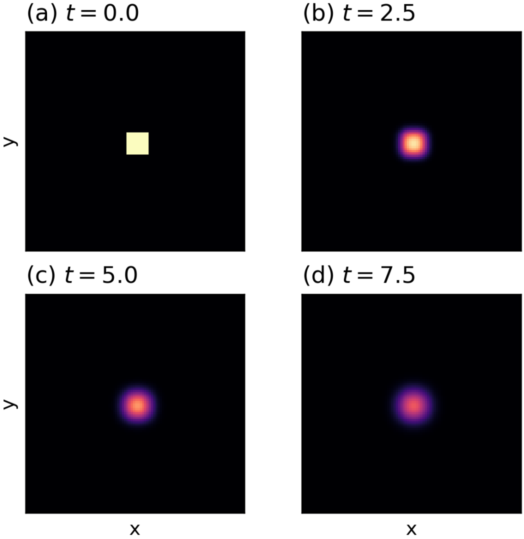
$$\frac{U_{i,j}^{t+1} - U_{i,j}^{t}}{\Delta t} = \alpha \left(\frac{U_{i,j+1}^{t} - 2U_{i,j}^{t} + U_{i,j-1}^{t}}{\Delta x^2} + \frac{U_{i+1,j}^{t} - 2U_{i,j}^{t} + U_{i-1,j}^{t}}{\Delta y^2} \right) \tag{3}$$

Numerical solution

Finally, we can solve for $U_{i,j}^{t+1}$:

$$U_{i,j}^{t+1} = U_{i,j}^{t} + \Delta t \alpha \left(\frac{U_{i,j+1}^{t} - 2U_{i,j}^{t} + U_{i,j-1}^{t}}{\Delta x^2} + \frac{U_{i+1,j}^{t} - 2U_{i,j}^{t} + U_{i-1,j}^{t}}{\Delta y^2} \right) \tag{4}$$

The 2D heat equation



Agenda

1. Overview

2. Computer architecture

3. Example: Solving the heat equation

4. **Runtime profiling**

5. Parallel computing

6. Summary

On premature optimization

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil.** Yet we should not pass up our opportunities in that critical 3%." — Donald Knuth

On premature optimization

To consider before optimizing your program:

- There's no point optimizing code that doesn't contribute significantly to the overall runtime
- Don't rely on your intuition; use a profiler to determine what are the critical parts of your code!
- Don't forget about usability, maintainability and readability!

Profiling Python programs

The built-in Python profiler(s)

- A built-in profiler for Python programs is provided by the `cProfile` module
- Profile function `my_function` from within script:

```
import cProfile
cProfle.run('my_function()')
```

- Profile whole script from command line:

```
$ python -m cProfile my_script.py
```

- There's also the `profile` module, which provides the same functionality but is slower for typical use cases.

Profiling Python programs

Using the profiler

```python
import cProfile
cProfile.run('calculate_heat(u)')
```

Output

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    2.600    2.600 <string>:1(<module>)
     1    0.000    0.000    2.600    2.600 heat_equation.py:44(calculate_heat)
   100    2.598    0.026    2.599    0.026 heat_equation.py:9(step)
     1    0.000    0.000    2.600    2.600 {built-in method builtins.exec}
   200    0.001    0.000    0.001    0.000 {built-in method numpy.zeros}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Profiling Python programs

Using a line profiler

- The built-in profilers show execution only on function level.
- Basically all time is spent in the step function (surprise)
- The alternative is to use a line-by-line profiler such as `pprof`:

```
$ pip install pprofile
$ pprofile --exclude-syspath heat_equation.py
```

Profiling Python programs

Example output — line profiler

```
Command line: heat_equation.py
Total duration: 42.0227s
File: heat_equation.py
File duration: 41.0692s (97.73%)

Line #|      Hits|         Time| Time per hit|      %|Source code
------+----------+-------------+-------------+-------+-----------
    ...
    13|       100|   0.00149155|  1.49155e-05|  0.00%|    l = np.zeros((n, n))
    14|      9900|    0.0351388|  3.54938e-06|  0.08%|    for i in range(1, n-1):
    15|    970200|      3.46738|  3.57388e-06|  8.25%|        for j in range(1, n-1):
    16|    960400|      3.66515|  3.81627e-06|  8.72%|            down = u[i - 1, j]
    17|    960400|      3.69363|  3.84593e-06|  8.79%|            up = u[i + 1, j]
    18|    960400|      3.70861|  3.86153e-06|  8.83%|            left = u[i, j - 1]
    19|    960400|      3.66988|   3.8212e-06|  8.73%|            right = u[i, j + 1]
    20|    960400|      3.65751|  3.80832e-06|  8.70%|            center = u[i, j]
    ...
```

- No single critical step.
- Rather, all steps in the inner loop seem to contribute equally.

Summary

- In more complex programs, profiling is necessary to determine where compute time is spent.
- Trade-off between overhead and accuracy.
- Python hides away a lot of low-level functionality, which may appear in profiling output and make it hard to interpret.

Exercise 2

- Exercise 2
- Time 10 minutes

Understanding Python performance

Interpreted languages (Python, JavaScript, . . .)

- Code is executed using an *interpreter*.
- The interpreter reads the source code and executes statements on-the-fly.

Compiled languages (C++, C, Fortran . . .)

- Code is compiled into an executable.
- The executable contains machine instructions that can be executed on the target hardware.

The CPython[1] execution model

- Python behaves like an interpreted language (no need for a compiler)
- But it actually does compile code to *byte code* (contained in the `__pycache__` folder)
- This code is executed on a virtual machine
- **Because of the VM, executing byte code is slower than directly executing machine code from a compiled executable.**

Libraries and C-extensions

- Modules can also be written in C and compiled into machine code.
- Python code can call functions from other libraries written in any other language. In this case performance is no different than calling it from any other language.

[1] CPython is the reference and most popular implementation of Python, but there exists others

What it means for performance

- Avoid deeply nested loops
- Instead, try to minimize the number of pure Python statements in your code
- A lot of packages call compiled C extensions or libraries under the hood (e.g. `numpy`). These will usually be fast.
- When performance matters, think of Python as a glue language to tie together calls to compiled and optimized libraries.

Agenda

1. Overview

2. Computer architecture

3. Example: Solving the heat equation

4. Runtime profiling

5. Parallel computing

6. Summary

Levels of parallelism

Processor level

- SIMD (Single instruction multiple data):
  - Modern CPUs can perform multiple addition/multiplication operations in a single cycle.

Thread level

- Also referred to as Shared-memory parallelism
- Organizing code in threads allows parts of the program to be executed in parallel.
- Shared memory makes communication between thread easy.

Process level

- Also referred to as distributed parallelism
- Processes may even run on different computers
  - This is needed to scale computations to modern cluster systems (Vera @ C3SE)

CPython: It's complicated

- The **global interpreter lock** (GIL): Only one thread may execute Python byte code at a time.
  - Python code essentially runs on a single-core VM.
- Need process-level parallelism to execute Python code in parallel
- However, IO-bound code can still benefit from thread-level parallelism.

Parallel computing in Python

Built-in tools: `threading.Threads`

- The `threading` module provides a low-level interface to execute code in separate threads.
- Tasks to execute in a thread are implemented as subclasses of the `Thread` class.

Example

```python
import threading
class MyThread(threading.Thread):
    def __init__(self):
        super().__init__()
    def run(self):
        name = self.name # or self.native_id in Python >= 3.8
        print(f"Hi from thread {name}")

for i in range(5):
    MyThread().start()
```

A more realistic example

```python
def get_temperature(lat, lon):
    """
    Get temperature for given location in Sweden from SMHI forecast.

    Args:
        lat(float): The latitude coordinate of the location.
        lon(float): The longitude coordinate of the location.

    Return:
        The temperature at the requested location as a float.
    """
    url = (f"https://opendata-download-metfcst.smhi.se/api/category/pmp3g/"
           f"version/2/geotype/point/lon/{lon}/lat/{lat}/data.json")
    response = urllib.request.urlopen(url)
    data = json.loads(response.read())
    forecast = data["timeSeries"][0]
    for parameter in forecast["parameters"]:
        if parameter["name"] == "t":
            temperature = parameter["values"][0]
    return temperature
```

Sequential version

```python
CITIES = [("Malmo" , (55.36, 13.02)),
          ("Goteborg", (57.42, 11.58)),
          ("Stockholm", (59.19, 18.4)),
          ("Umea" , (63.49, 20.15)),
          ("Lulea", (65.35, 22.9))]

def get_temperatures_sequential():
    for city, coords in CITIES:
        temperature = get_temperature(*coords)
        print(f"The temperature in {city} is {temperature} deg. C.")
```

Output

- Required time: 872 ms

```
The temperature in Malmo is 11.8 deg. C.
The temperature in Goteborg is 11.3 deg. C.
The temperature in Stockholm is 8.6 deg. C.
The temperature in Umea is 7.5 deg. C.
The temperature in Lulea is 4.8 deg. C.
```

Concurrent version

```python
class TemperatureGetter(threading.Thread):
    """Thread class to request temperature."""
    def __init__(self, city, coords):
        super().__init__()
        self.city = city
        self.coords = coords

    def run(self):
        self.temperature = get_temperature(*self.coords)

def get_temperatures_threads():
    threads = []
    for city, coords in CITIES:
        thread = TemperatureGetter(city, coords)
        thread.start()
        threads.append(thread)

    for thread in threads:
        thread.join() # Wait for thread to finish execution

    for thread in threads:
        city = thread.city
        temperature = thread.temperature
        print(f"The temperature in {city} is {temperature} deg. C.")
```

Output

- Required time: 286 ms

```
The temperature in Malmo is 11.8 deg. C.
The temperature in Goteborg is 11.3 deg. C.
The temperature in Stockholm is 8.6 deg. C.
The temperature in Umea is 7.5 deg. C.
The temperature in Lulea is 4.8 deg. C.
```

Result

- The threaded version is much faster although no code is executed in parallel (remember the GIL).
- This is because a lot of the required time is spent waiting for the result from SMHI and does not require any computing to be performed.

Built-in tools: `concurrent.futures`

- The `concurrent` module provides a more elegant way of writing concurrent code.
- Tasks can be created directly by passing a function to an executor object.
- Depending on whether `ThreadPoolExecutor` or `ProcessPoolExecutor` is used the code is run in separate threads or processes.

```python
from concurrent.futures import ThreadPoolExecutor

def say_hi(i):
    print(f"Hi from task {i}!")

executor = ThreadPoolExecutor(max_workers=5)
results = []
for i in range(5):
    results.append(executor.submit(say_hi, i))
```

The `get_temperature` example

```python
from concurrent.futures import ThreadPoolExecutor

def get_temperatures_thread_pool():
    executor = ThreadPoolExecutor(max_workers=5)
    results = []
    for city, coords in CITIES:
        results.append(executor.submit(get_temperature, *coords))
    for (city, _), result in zip(CITIES, results):
        temperature = result.result()
        print(f"The temperature in {city} is {temperature} deg. C.")
```

Notes

- The `Executor.submit` method returns a `future` object, which can be queried for the result and state of the computation.

Exercise 3

- Exercise 3 in notebook
- Time: 15 minutes

Exercise 3

Results

- Because of the GIL, using parallelization using threads yields no benefits for compute-bound tasks.

Parallel computing in Python

Built-in tools: `coroutines`

- Coroutines are defined using the `async` keyword.
- Calling a coroutine immediately returns a task object (somewhat similar to `Executor.submit`)
- A coroutine can wait for another coroutine using the `await` keyword.

Example

```python
import asyncio

async def say_hi():
    print("hi")

task = say_hi()
print("waiting ...")  # Prints: waiting ...
asyncio.run(say_hi()) # Prints: hi
```

The `get_temperature` example using coroutines

```python
import aiohttp
import asyncio

async def get_temperature(session, lat, lon):
    url = (f"https://opendata-download-metfcst.smhi.se/api/category/pmp3g/"
          f"version/2/geotype/point/lon/{lon}/lat/{lat}/data.json")
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            data = json.loads(await response.read())
            forecast = data["timeSeries"][0]
            for parameter in forecast["parameters"]:
                if parameter["name"] == "t":
                    temperature = parameter["values"][0]
    return temperature

CITIES = [("Malmo" , (55.36, 13.02)),
          ("Goteborg", (57.42, 11.58)),
          ("Stockholm", (59.19, 18.4)),
          ("Umea" , (63.49, 20.15)),
          ("Lulea", (65.35, 22.9))]

async def print_temperatures():
    session = aiohttp.ClientSession()
    tasks = [get_temperature(session, *coords) for _, coords in CITIES]
    for (city, _), task in zip(CITIES, tasks):
        temperature = await task
        print(f"The temperature in {city} is {temperature} deg. C.")
    session.close()

asyncio.run(print_temperatures())
```

Comments on coroutines

- Still a relatively new Python feature
- Mostly targeted at IO-limited tasks (server applications)

How can we parallelize the heat equation?

- Compute-limited problem: Can’t use threads in pure Python
- Process-based parallelization will require inter-process communication (remember: no shared memory)
- Difficult solutions:
  - Writing a C-extension and using multithreading
  - Using MPI

Parallel computing in Python

The best of both worlds: Numba

- `numba` is a Python package that let's you compile specific Python functions.
- It can automatically parallelize your applications.
- Just in time compilation (JIT): Functions are compiled the first time they are used.

Example

```python
from numba import jit

@jit(nopython=True)
def sum(x):
    result = 0.0
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            result += x[i, j]
```

Parallel example

- To tell numba to try to parallelize a loop use prange instead of the built-in range function:

```
from numba import jit, prange

@jit(nopython=True, parallel=True)
def sum(x):
    result = 0.0
    for i in prange(x.shape[0]):
        for j in range(x.shape[1]):
            result += x[i, j]
```

Timing results

```
numba (Serial):   6.05 ms
numba (Parallel): 2.36 ms
numpy (Serial):   1.79 ms
```

Exercise 4

- Exercise 3 in notebook
- Time 15 minutes

Agenda

CHALMERS
UNIVERSITY OF TECHNOLOGY

1. Overview

2. Computer architecture

3. Example: Solving the heat equation

4. Runtime profiling

5. Parallel computing

6. **Summary**

Summary

- Compared to a compiled language, pure Python is slow
- But Python can still be fast for:
  - IO-bound problems.
  - When functions from a C-extension or library are called.
  - When you use the right tools (numba, jax, torch, tensorflow . . .)