Parallel Processing

Modern Computational Methods in Physics 2.

Balázs *PÁL*

[1]Eötvös Loránd University
[2]Wigner Research Centre for Physics

Topics

Questions

- „How can I run this code in parallel?"
- „What should I use in Python to run my code in parallel?"
- „Can I run a Jupyter Notebook in parallel somehow?"
- „Does Python code run in parallel by default?"

⋮

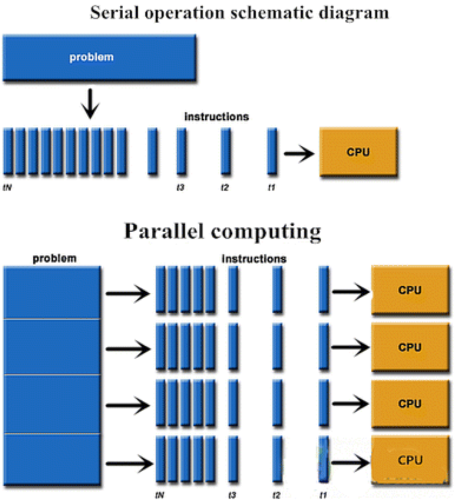
Answers

- No. Just no. Please no. No.
- How much is behind „parallelisation"?
- When and why are we even using it?
- In what situations and how can a physicist use it?

Overview

Important concepts

- Fundamental terms
  - Thread
  - Process
  - Core
- Computational methods
  - Serial/Sequential
  - Parallel
- Parallelisation methods
  - Multi-threading
  - Multi-processing
  - ⋮



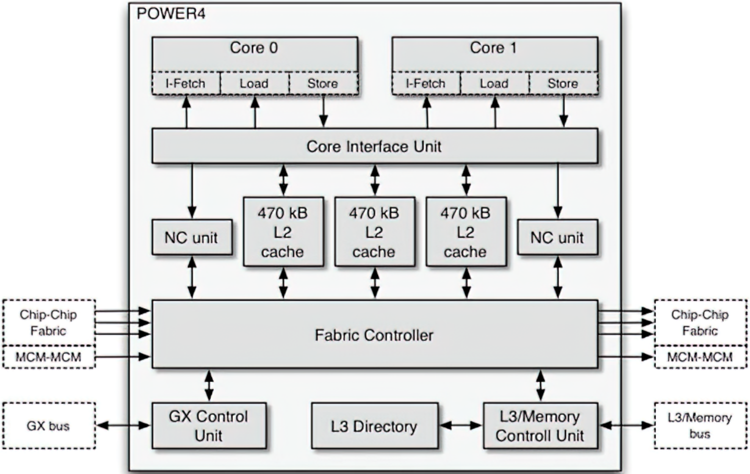
Source: ResearchGate

Threads, Processes, Cores and other creatures

CPU („Central Processing Unit”) and cores

- „The brain of a computer”, it is responsible for
  - the calculation of arithmetic operations,
  - performing logical operations,
  - operation of the other hardware components.
- Nowadays all CPUs are multi-core
  - First multi(two)-core model: POWER4 (IBM, 2001)

Threads, Processes, Cores and other creatures



Source: *ibm.com*

Threads, Processes, Cores and other creatures

CPU („Central Processing Unit”) and cores

- „The brain of a computer”, it is responsible for
  - the calculation of arithmetic operations,
  - performing logical operations,
  - operation of the other hardware components.
- Nowadays all CPUs are multi-core
  - First multi(two)-core model: POWER4 (IBM, 2001)

Processes, Threads

- Process
  - Name for a single, running program
- Thread
  - A running sequence of operations
  - A process can be broken down into several threads
  - They always run „in parallel”

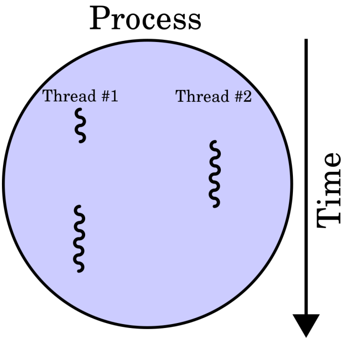
Running Threads

Managing threads

1. On the application level
   - It is up to the user to decide which part of the program to split into threads and how
2. On the OS level
   - The OS thread manager decides the order at which threads are run

Running in parallel

- With **more cores** than threads, they actually run in parallel
- With **more threads** than cores, the CPU periodically cycles through them



Source: Cburnett, Wikipedia

An example for a parallelisable problem

Cosmological N-body simulations

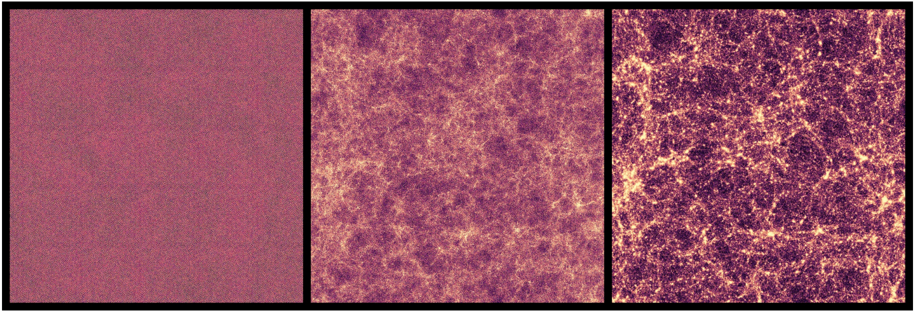
Lots and lots of similar calculations...

$$\Psi\left(\boldsymbol{q},\tau\right)=\int\frac{\mathrm{d}^3\,k}{\left(2\pi\right)^3}e^{i\boldsymbol{k}\cdot\boldsymbol{q}}\frac{i\boldsymbol{k}}{k^2}\delta_L\left(\boldsymbol{k}\right)$$

$$\boldsymbol{g}_i=-\nabla\varphi\left(\boldsymbol{x}_i\right)=G\sum_{\substack{j\neq i\\ j=1}}^{N}m_j\mathcal{F}\left(\boldsymbol{x}_i-\boldsymbol{x}_j\right)$$

$$\boldsymbol{x}=\boldsymbol{q}+\Psi\left(\boldsymbol{q},\tau\right)$$

$$\dot{\boldsymbol{x}}=\frac{\dot{D}\left(a\right)}{D\left(1\right)}\Psi\left(\boldsymbol{q},\tau\right)$$



But there are many more...

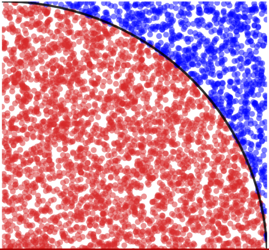
Examples of parallelizable problems

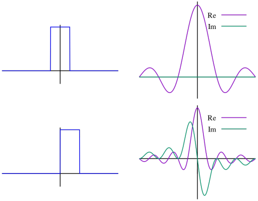
- Monte–Carlo-simulation
- Numerical integration
- Machine learning algorithms (e.g. random forest)

  ⋮

- Computer visualisations and modelling
- Problems related to image processing
- Discrete Fourier transform

  ⋮





Race condition

Parallelisation in practice

Several tools now make programming parallel code much easier:

- Python: `threading`, `multiprocessing` etc.
- C++17: Standard libraries
- C/C++/Fortran: `OpenMP`, `CUDA`, etc.

Race condition

Parallelisation in practice

Several tools now make programming parallel code much easier:

- Python: `threading`, `multiprocessing` etc.
- C++17: Standard libraries
- C/C++/Fortran: `OpenMP`, `CUDA`, etc.

Issues that needs to be addressed

- There would be plenty without the tools mentioned above, but there's still many more...
- Probably the most important one is „race condition"
  - Two threads want to access the same memory space at the same time
  - A typical case of the dreaded „undefined behaviour"
  - A critical bug that can completely mess up the behaviour of any running process

Race condition

A simple (but kinda dumb) example

race-condition-serial.cpp

```cpp
int addition(int &x) {

  int new_x = x + 1;

  return new_x;
}

int main(int argc, char const *argv[])
{
  // Starting number
  int x = 0;

  // Adding +1 to it 4 times
  for(int i = 0; i < 4; i++) {
    x = addition(x);
    std::cout << "Current value of `x` is " << x << std::endl;
  }

  return 0;
}
```

```
└─$ ./race_condition_serial
Current value of `x` is 1
Current value of `x` is 2
Current value of `x` is 3
Current value of `x` is 4
```

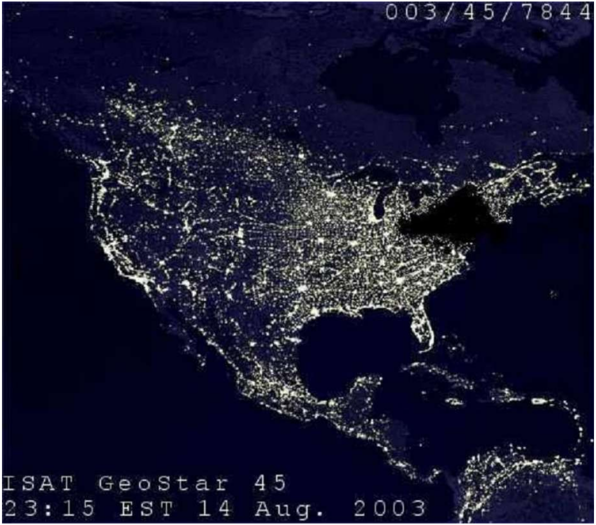
race-condition-parallel.cpp

```cpp
int addition(int x)
{
  int new_x = x + 1;

  return new_x;
}

int main(int argc, char const *argv[])
{
  // Starting number
  int x = 0;

  // Parallelization
  std::vector<std::future<int>> future_vec;

  // Adding +1 to it 4 times
  for(int i = 0; i < 4; i++)
  {
    future_vec.push_back(std::async(std::launch::async, addition, x));
  }

  for(int i = 0; i < 4; i++)
  {
    auto new_x = future_vec[i].get();
    std::cout << "Current value of `x` is " << new_x << std::endl;
  }

  return 0;
}
```

```
└─$ ./race_condition_parallel
Current value of `x` is 1
Current value of `x` is 1
Current value of `x` is 1
Current value of `x` is 1
```

Race condition

Northeast blackout of 2003



Source: clevescene.com

Amdahl's law

Motivation

How much time can be saved by parallelisation in real cases?

Amdahl's law

Motivation

How much time can be saved by parallelisation in real cases?

- The runtime of a program can be expressed as follows:

$$T = T \cdot S + T \cdot P$$

where $S + P = 1$ and which are usually the ratio of the number of parts of an algorithm that can be run only in a serial manner (S) and the number of parts that can be parallelized (P).

Amdahl's law

Motivation

How much time can be saved by parallelisation in real cases?

- The runtime of a program can be expressed as follows:

$$T = T \cdot S + T \cdot P$$

where $S + P = 1$ and which are usually the ratio of the number of parts of an algorithm that can be run only in a serial manner (S) and the number of parts that can be parallelized (P).

- If the parallelizable part (P) is divided into several (N) threads, the runtime of the program is reduced:

$$T_{\mathrm{new}} = T \cdot S + T \cdot \frac{P}{N}$$

Amdahl's law

Motivation

How much time can be saved by parallelisation in real cases?

- The speedup (Q) now can be expressed as:

$$Q\left(N\right) = \frac{T}{T_{\text{new}}} = \frac{\cancel{T}}{\cancel{T} \cdot S + \cancel{T} \cdot \frac{P}{N}} = \frac{1}{S + \frac{P}{N}}$$

Amdahl's law

Motivation

How much time can be saved by parallelisation in real cases?

- The speedup (Q) now can be expressed as:

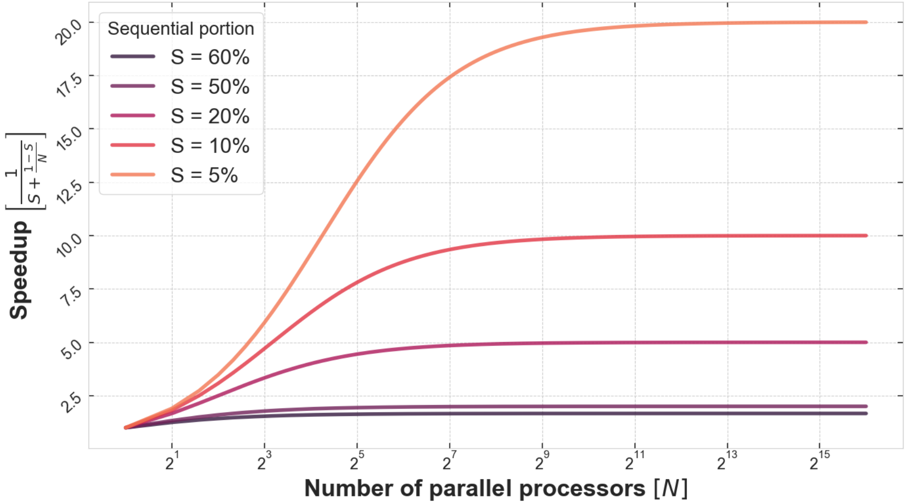
$$Q\left(N\right) = \frac{T}{T_{\mathrm{new}}} = \frac{\cancel{T}}{\cancel{T} \cdot S + \cancel{T} \cdot \frac{P}{N}} = \frac{1}{S + \frac{P}{N}}$$

- After some simplification, we get back the usual form of Amdahl's law:

$$Q\left(S, N\right) = \frac{1}{S + \frac{P}{N}} = \frac{1}{S + \frac{1-S}{N}},$$

which specifies that the speedup depends solely on the proportion of parts to be run in series and the number of threads.

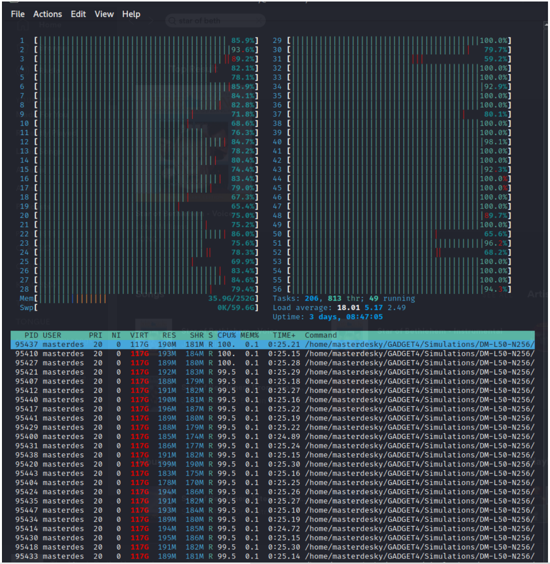
Amdahl's law


Sequential portion
S = 60%
S = 50%
S = 20%
S = 10%
S = 5%
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
Speedup $\left[\frac{1}{S+\frac{1-S}{N}}\right]$
2^1
2^3
2^5
2^7
2^9
2^{11}
2^{13}
2^{15}
Number of parallel processors $[N]$

Parallelisation in physics

In general

- Running simulation software (e.g. Computer Simulation and Modelling, MSc course – OpenFoam, GADGET4, HOOMD-blue)
- Working on servers (e.g. onco2, tesla, atys, veo1 (known as ‘kooplex‘) etc. at ELTE)
- Working on HPC clusters (e.g. ELTE *Atlasz* cluster or the Hungarian Komondor HPC cluster, etc.)



```
#!/bin/bash -l
#SBATCH --job-name="G4M2Norm"
#SBATCH --partition=hpc2019
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=18
#SBATCH --cpus-per-task=1
#SBATCH --time=24:00:00
#SBATCH --exclusive

echo
echo "Running on hosts: $SLURM_NODELIST"
echo "Running on $SLURM_NNODES nodes."
echo "Running on $SLURM_NPROCS processors."
echo

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/users/lordpb666/opt/gsl-2.6/lib/
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/users/lordpb666/opt/fftw-3.3.9/lib/

mpirun -np $SLURM_NPROCS /users/lordpb666/apps/GADGET4/Gadget4 \
                         /users/lordpb666/apps/GADGET4/Simulations/DM-N65536-M2-L200-GFT5/GADGET4_GTF5.param
```

```
          JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
         314254   hpc2019  7-28-61 hanyeczo  R 23-13:29:58      1 cn18-25
         314255   hpc2019  7-62-95 hanyeczo  R 23-13:29:52      1 cn18-26
         314256   hpc2019 7-96-128 hanyeczo  R 23-13:29:49      1 cn18-07
         314257   hpc2019  14-1-42 hanyeczo  R 23-13:26:30      1 cn18-08
         314258   hpc2019 14-43-84 hanyeczo  R 23-13:25:27      1 cn18-09
         314259   hpc2019 14-85-12 hanyeczo  R 23-13:24:23      1 cn18-10
       316125_8   hpc2019 evo22.sb skiszkao  R 12-10:50:51      1 cn18-24
         325390   hpc2019 atlasz_b geobarbi  R 5-06:35:54      5 cn18-[04,11-12,16,18]
         326521   hpc2019 atlasz_b geobarbi  R 5-12:35:29      1 cn18-06
         326839   hpc2019 atlasz_b geobarbi  R 5-06:07:18      1 cn18-35
         327902   hpc2019 atlasz_b geobarbi  R 4-04:12:25      2 cn18-[05,27]
         327919   hpc2019 atlasz_b geobarbi  R 3-16:49:20      1 cn18-33
         327942   hpc2019 atlasz_b geobarbi  R 3-10:09:39      1 cn18-03
```

Parallelisation in physics

Using Python

Prerequisites

- E.g. the `threading`, `multiprocessing`, `joblib`, `subprocess` etc. packages mentioned in the previous section
- Multiple CPU cores available for ordering (e.g. this is 2 in the case of Kooplex)

```
_ = Parallel(n_jobs=n_jobs)(delayed(create_frame)(i, P,
                                                  N, n_steps, gl,
                                                  outdir, ffmt, fdpi) for i, gl in enumerate(tqdm(grid_lims)))
```

```
# Takes lot of time to write all files and ping all sites!
for target in df_n['URL']:
    ping_command = 'ping -D -c {0} -i {1} -s {2} {3}'.format(n_packet, interval, packet_size, target)
    output = "{0}{1}.txt".format(data, target)
    with open(output, 'w') as f:
        # Using `Popen` here to run pings "parallel"
        #print('Pinging {}...'.format(target))
        process = subprocess.Popen(ping_command.split(' '), stdout=f)
```