Computação Paralela / Computação Avançada

2022–2023 1st semester

Helmut Wolters
helmut@coimbra.lip.pt

LIP / Universidade de Coimbra

cap1 — 2022-09-21

The Idea of Parallel Programming

```
for i in range(1000):
  perform one step of a calculation
  save somewhere
Process final result from iterations
```

- Academical example — roll dice:

```
from random import randint
D = {}
for i in range(1, 7):
    D[i] = 0
for i in range(1000):
    v = randint(1, 6)
    D[v] += 1
print (D)
```

The Idea of Parallel Programming

- If we have 1000 processors, we can speed up the calculation by giving one iteration to each processor and have them do the calculation all at the same time.
- But this is only possible, if the iterations are independent of one another.
- And this makes only sense, if each iteration takes roughly the same amount of time.
- And then we need a way to share the results of each iteration.
- We need a kind of messaging system between the involved processors.
- After having done its calculation, each processor sends the result to some central node;
- The central node then continues the processing of the results, or starts a new distributed process.

Message Passing Interface — MPI

- The current *de facto* standard for message passing is the so called *Message Passing Interface (MPI)*.
- MPI is not an IEEE or ISO standard, but has in fact become the "industry standard" for writing message passing programs on HPC platforms.
- MPI exists for many programming languages, among many others for *Python*, *C++* and *Fortran*.
- Several implementations do exist
- *Open MPI* is an open source MPI implementation developed and supported by a consortium of academic, research, and industry partners.
- On *Python*, *Open MPI* is implemented through a module called *mpi4py* .

Message Passing Interface — MPI

- In MPI, there is no "central" node.
- An arbitrary number of n processes work in parallel, and can communicate between each other by sending and/or receiving messages.
- All processes are serially numbered with a so-called *rank*, which is just a number and defines no hierarchy.
- As a matter of choice we will use the process with rank 0 as the one that collects the data of all other processes and performs the final evaluation.
- But the process 0 will also perform one thread of the calculus itself.
- We will use one single source code, where at a certain point we will have two branches, one for process 0, and one for all the others.
- But the code for the calculus to run in parallel will be equal for all processes, including process 0.

Message Passing Interface — MPI

- Although the idea behind is to use High Performance Computing on a big machine with many processors for parallel computing, we can use a simple PC:
- Modern PCs have more than one CPU core
- A modern laptop has typically 2 or 4 physical cores, each of them with 2 "threads". Each thread works by hardware design of the CPU chip as if it was an independent processor.
- You can check the number of threads of your laptop:
  - On Linux: `cat /proc/cpuinfo | grep ^processor`
  - On Windows 10: `Ctrl-Shift-ESC` → Performance → "Cores:"
- Moreover, any multitask operating system can run thousands of processes virtually at the same time, by assigning timeslots to each process and thus executing them as if they were running "in parallel".
- So for test purposes, we can run more processes in parallel than the number of our system's "real" processors
- But be careful not to exaggerate...

Message Passing Interface — MPI

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    print ('Number of processes:', MPI.COMM_WORLD.Get_size())
    for p in range(1, MPI.COMM_WORLD.Get_size()):
        data = comm.recv(source=p)
        print('On process', p, 'data is', data)
else:
    data = {'rank': rank, 'value': 10*rank}
    comm.send(data, dest=0)
```

Message Passing Interface — MPI

- Let's run this code using 4 threads from a shell:

```
# mpirun -n 4 python mpi1.py
Number of processes: 4
On process 1 data is {'rank': 1, 'value': 10}
On process 2 data is {'rank': 2, 'value': 20}
On process 3 data is {'rank': 3, 'value': 30}
```

- You can run as many threads in parallel as you like.
- But if you exaggerate, the system will freeze due to overload of RAM.
- Obviously, for "real" parallel processing, you would not run more threads than your system has cores.
- This can be several cores of a normal PC,
  or the processor array of a supercomputer.

Example - Roll the Dice

- To see how we can do a simulation, we will simulate rolling the dice through random numbers
- The "result" of the simulation is the frequency distribution of the number of events for each face of the dice
- Random numbers in Python:
  - equally distributed float random number between 0 and 1:

```
from random import random
x = random()
```

  - equally distributed integer random number between j and k:

```
from random import randint
x = randint(j, k)
```

Example - Roll the Dice

- Examples in `cap1-examples.zip`:

dice1.py single core simulation of rolling the dice n times

dice2.py parallel simulation where each thread rolls one die once
$\rightarrow$ many threads (n)

dice3.py parallel simulation where each thread takes over an equal fraction of rolling the dice: n/number of threads

- Every program has small variants modifying the total number of rolls and for different numbers of parallel threads.
(`dice3a.py, dice3b.py, ...`)