Distributed Computing and Introduction to High Performance Computing

Imad Kissami[1]

[1]Mohammed VI Polytechnic University, Benguerir, Morocco

May 10, 2023

OUTLINE

- Why HPC?
- What’s Supercomputer?
- Data locality
- How to make Python Faster
- Parallel models s
- Performance metrics

OUTLINE

- The flood of Data
- Big data problem
- What’s HPC ?
- Typical HPC workloads
- Data Analytics Process

THE FLOOD OF DATA

In 2021

- Internet user ~ 1.9 GB per day
- Self driving car ~ 4 TB per day
- Connected airplane ~ 5TB per day
- Smart factory ~ 1 PB per day
- Cloud video providers ~ 750 PB per day

THE FLOOD OF DATA

A self-driving car

- Radar ~ 10 – 100 KB per second
- Sonar ~ 10 – 100 KB per second
- GPS ~ 50 KB per second
- Lidar ~ 10 – 70 MB per second
- Cameras ~ 20 – 40 MB per second
- 1 car ~ 5 Exaflops per hour

BIG DATA PROBLEM

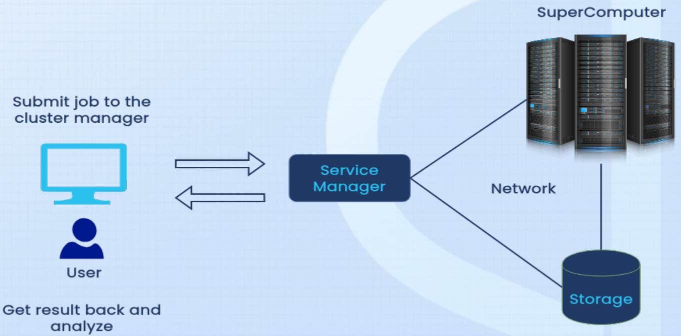
Too much data

Not enough computer power, storage or infrastructure

WHAT'S HPC?

Leveraging distributed compute resources to solve complex problems

- Terabytes ⟶ Petabytes ⟶ Zetabytes of data
- Results in minutes to hours instead of days or weeks

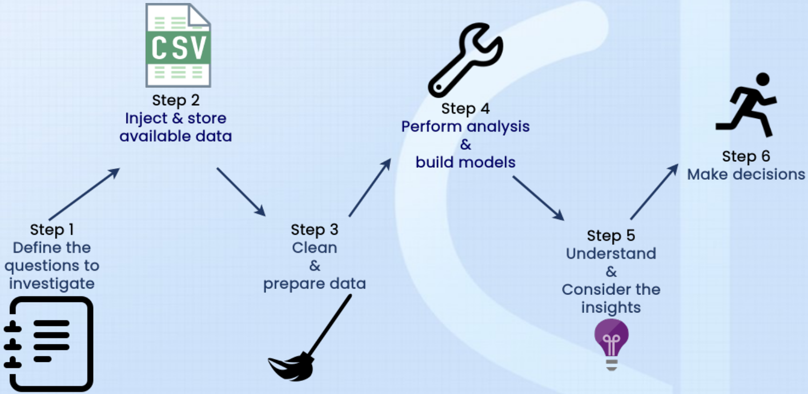


TYPICAL HPC WORKLOADS

* Source:
https://www.xilinx.com/applications/data-center/high-performance-computing.html

DATA ANALYTICS PROCESS

Inspecting, cleaning, transforming and modeling ➠ decision-making.



SUMMARY

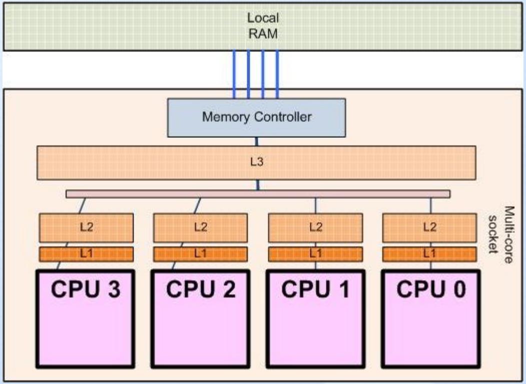
- Larger datasets require distributed computing
- Several open source HPC frameworks available

OUTLINE

- A brief introduction on hardware
- Modern supercomputers

A BRIEF INTRODUCTION ON HARDWARE

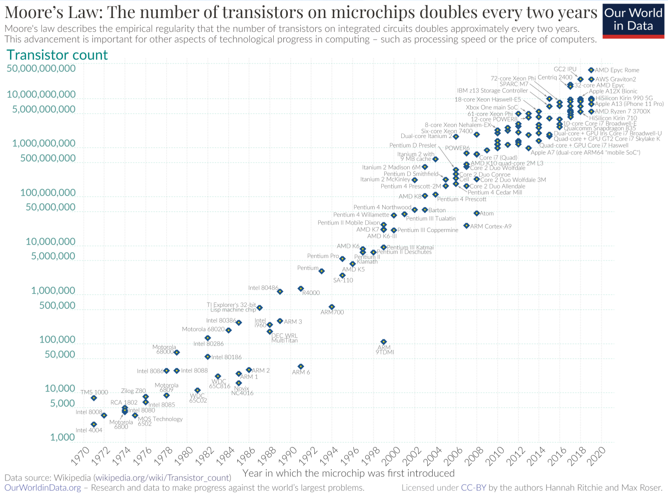
Modern architecture (CPU)



A BRIEF INTRODUCTION ON HARDWARE

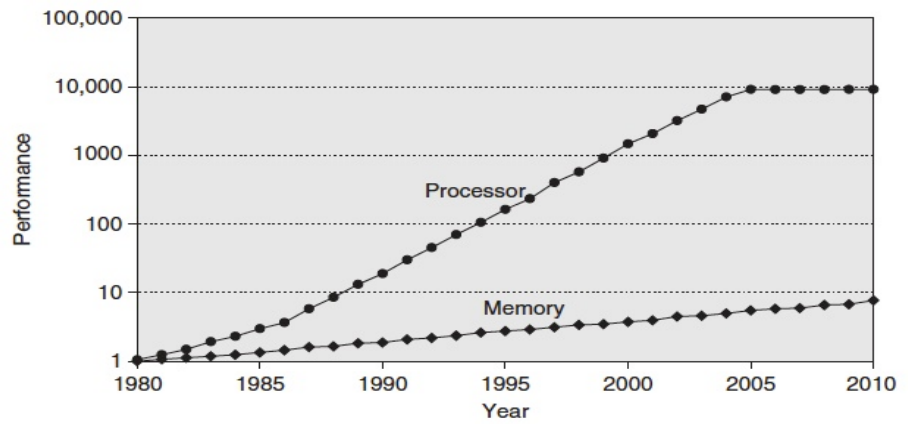
Moore's Law

- Number of transistors: from 37.5 million(2000) to fifty billion(2022)
- Cpu speed: from 1.3GHz to 3.4GHz



A BRIEF INTRODUCTION ON HARDWARE

CPU vs RAM speeds



Common Processors

Processor	Launched	Nb. of Cores	Freq. (Ghz)
Xeon Platinum 9282 (formerly Cascade Lake)	2019-Q2	28	2.6-3.8
Xeon Platinum 8376H (formerly Cooper Lake)	2019-Q2	28	2.6-4.3
i9-12900H (Mobile, 12th generation)	2022-Q1	4-16	3.8-5.0
i9-12900KS (Desktop, formerly Alder Lake)	2022-Q1	8-16	2.5-5.5

Table: Some Intel processors

Processor	L3 cache	Nb. of Cores	Freq. (Ghz)
AMD EPYC 7773X	768 MB	64	2.2-3.5
AMD EPYC 7763	256 MB	64	2.45-3.5
AMD Ryzen 9 5950X (Desktop)	72 MB	16-32	3.4-4.9
AMD Ryzen 9 3900X (Desktop)	70 MB	12-24	3.4-4.6

Table: Some AMD processors

What is a supercomputer?

- cdc 6600: 1964 - three million calculations per second
- Summit: 2018 - 36000 processors - 200 quadrillion calculations per second
- Frontier: 2022 - 8 million processors - AMD EPYC with 64 cores and speed up to 2GHz - quintillion calculations per second
  Toubkal: 2021 - 69000 processors

MODERN SUPERCOMPUTERS

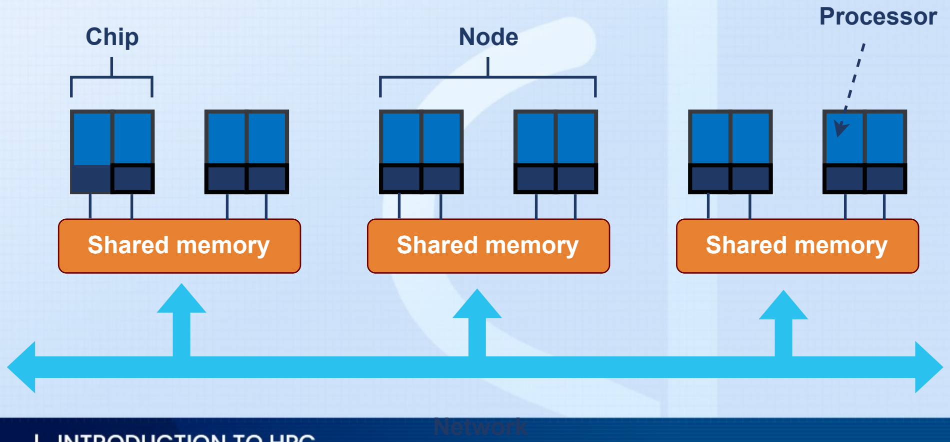
What is a supercomputer?

Frontier (USA)

MODERN SUPERCOMPUTERS

What is a supercomputer?

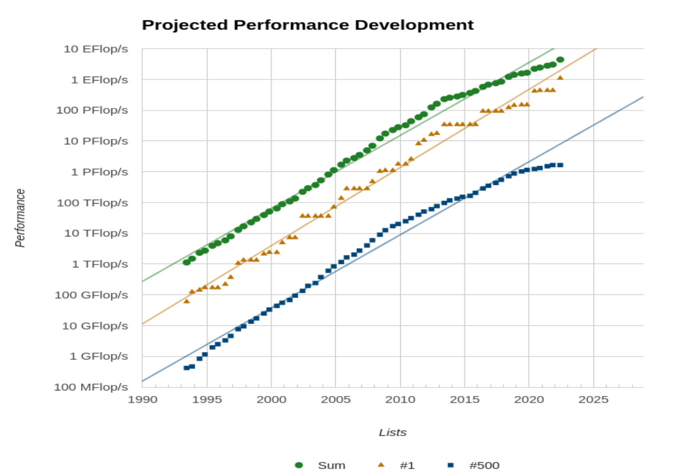
Cluster



MODERN SUPERCOMPUTERS

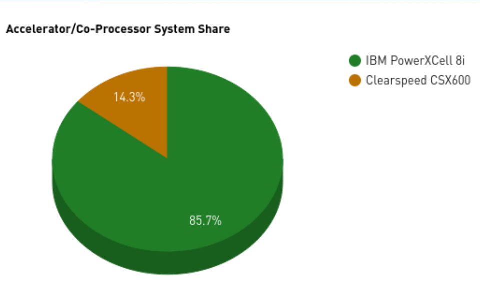
Top 500

- Cray 2: Gigascale milestone in 1985
- Intel ASCI Red System: Terascale in 1997
- IBM Roadrunner System: Petascale in 2008
- Frontier: Exascale in 2022



Modern supercomputers

Top 500 Family system share evolution November 2009

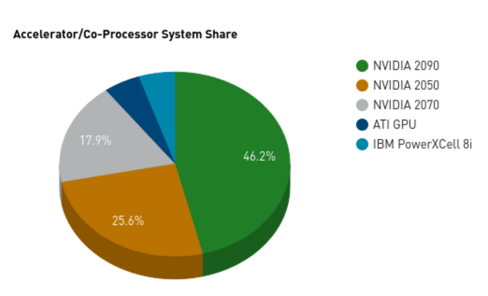


[1]

[1] https://www.top500.org/statistics/list/

Modern supercomputers

Top 500 Family system share evolution November 2011

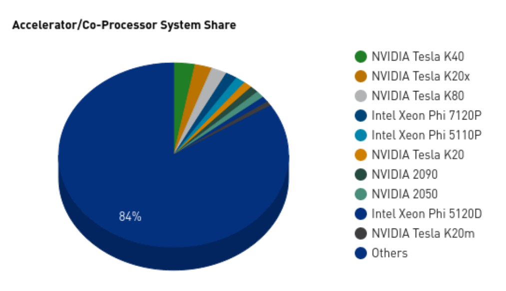


1

[1] https://www.top500.org/statistics/list/

Modern supercomputers

Top 500 Family system share evolution November 2015

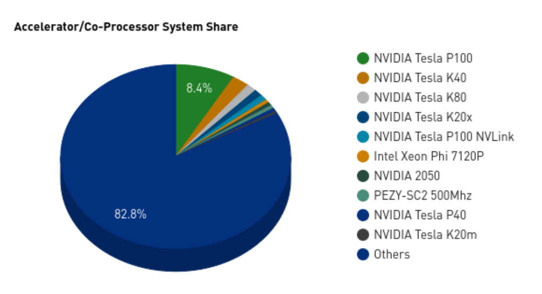


1

[1] https://www.top500.org/statistics/list/

Modern supercomputers

Top 500 Family system share evolution November 2017



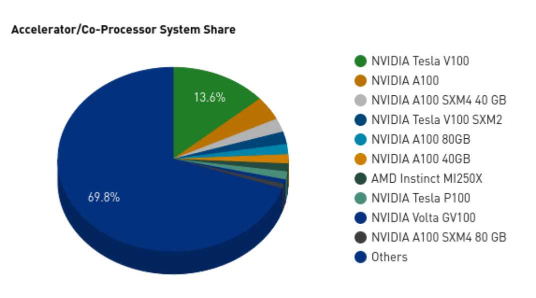
1

[1] https://www.top500.org/statistics/list/

Modern supercomputers

Top 500 Family system share evolution

June 2022



1

[1] https://www.top500.org/statistics/list/

SUMMARY

- Highlights
  - New architectures are available
  - Supercomputers achieve Exascale
- Consequence for the developers
  - Writing dedicated codes

OUTLINE

- Some definitions
  - FLOPS
  - Frequency
  - Memory Bandwidth
  - Memory Latency
- Computational Intensity
- Two level memory model

SOME DEFINITIONS

FLOPS

Floating point operations per second (FLOPS or flop/second).

SOME DEFINITIONS

Frequency

Speed at which a processor or other component operates (Hz)

SOME DEFINITIONS

Memory Bandwidth

Rate at which data can be transferred between the CPU and the memory (bytes/second).

SOME DEFINITIONS

Memory Latency

Time delay between a processor requesting data from memory and the moment that the data is available for use (clock cycles or time units).

COMPUTATIONAL INTENSITY

Algorithms have two costs (measured in time or energy):

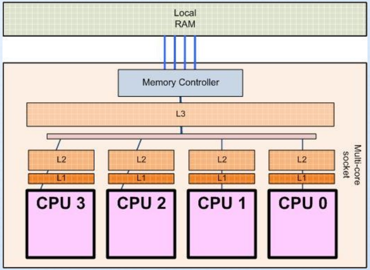
- Arithmetic (FLOPS)
- Communication: moving data between
  - levels of a memory hierarchy (sequential case)
  - processors over a network (parallel case)

Computational Intensity

It is the ratio between arithmetic complexity (or cost) and memory complexity (cost).

TWO LEVEL MEMORY MODEL

Modern architecture (CPU)



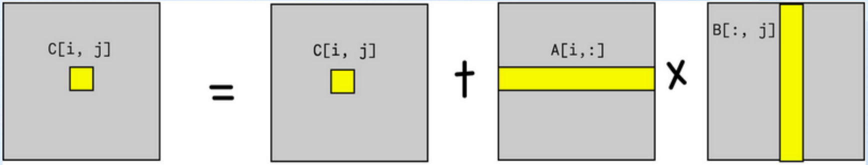
Cache Hit or Miss

- `Cache Hit`: CPU is able to find the Data in `L1/L2/L3`
- `Cache Miss`: CPU is not able to find the Data in `L1-L2-L3` and must retrieve it from `RAM`

Typical sizes

- `RAM` ~ 4 GB – 128 GB even higher on servers
- `L3` ~ 4 MB – 50 MB
- `L2` ~ 256 KB – 8 MB
- ➠ Holds data that is likely to be accessed by the CPU
- `L1` ~ 256 KB
- ➠ Instruction and Data cache

MATRIX MULTIPLICATION: THREE NESTED LOOP




```
for i in range(0, n):
    #read row i of A into fast memory
    for j in range(0, n):
    #read row C[i,j] into fast memory
    #read col j of B into fast memory
        for k in range(0, n):
            C[i,j] = C[i,j] + A[i,k]*B[k,j]
            #write C[i,j] back to slow memory
```

```
arithmetic cost         :: n**3*(ADD + MUL) = 2n**3 arithmetic operations
memory cost             :: n**3*READ + n**2*READ + n**2*(READ + WRITE) = n**3 + 3n**2
computational intensity :: 2n**3/(n**3 + 3n**2 ) ~= 2
```

SUMMARY

- Running time of an algorithm is sum of 3 terms:
  - `N_flops * time_per_flop`
  - `N_words / bandwidth`
  - `N_messages * latency`

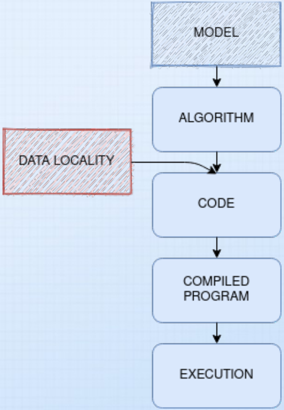
➠ Avoiding communication algorithms come with a significant speedup

- Some examples
  - Up to 12x faster for 2.5D matmul on 64K core IBM BG/P
  - Up to 3x faster for tensor contractions on 2K core Cray XE/6
  - Up to 6.2x faster for All-Pairs-Shortest-Path on 24K core Cray CE6

OUTLINE

- Data Locality
  - The Penalty of Stride
  - High Dimensional Arrays
- Block Matrix Multiplication

DATA LOCALITY



- Data locality is key for improving per-core performance,
- Memory hierarchy has 4 levels,
- Processor looks for needed data in memory hierarchy,
- Simple or complex manipulations can increase speedup,
- Blocking version of `mxm` can increase computational intensity.

The Penalty of Stride > 1?

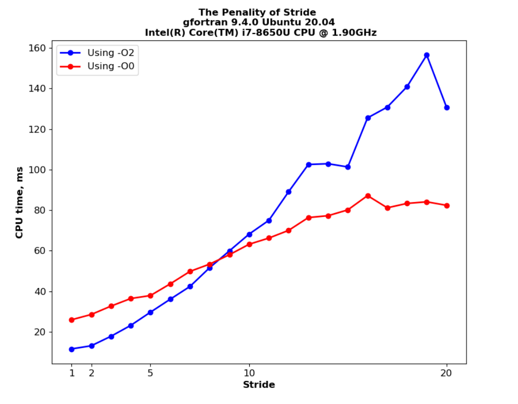
- Data should be arranged for unit stride access,
- Not doing so can result in severe performance penalty

Example:

```
do i=1, N*i_stride,i_stride
    mean = mean + a(i)
end do
```

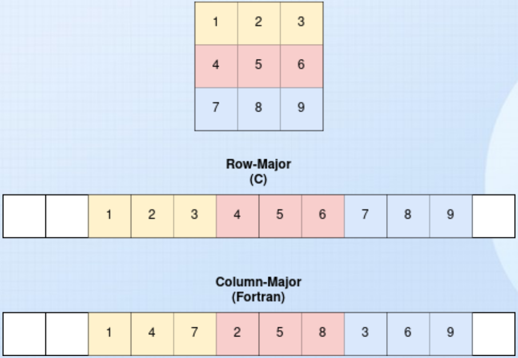
- Compilation with all optimization and vectorization disabled (-O0)
- Compulation with (-O2) that activates some optimizations

The Penalty of Stride: CPU time



DATA LOCALITY

High Dimensional Arrays



- High Dimensional Arrays are stored as a contiguous sequence of elements
- `Fortran` uses Column-Major ordering
- `C` uses Row-Major ordering

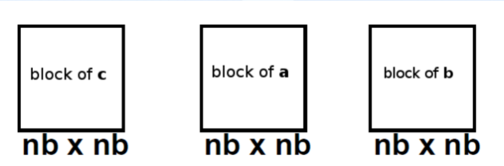
mxm in Fortran N = 1000

- Naive version: CPU-time 1660.6 (msec)
- Transpose version: CPU-time 1139.8 (msec)

BLOCK MATRIX MULTIPLICATION

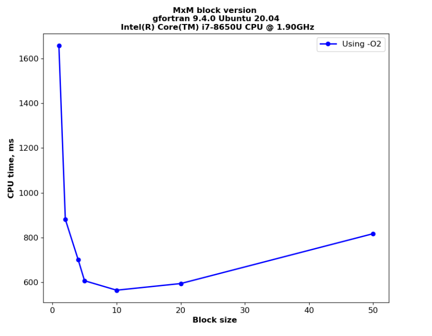
mxm example: Using block version (cache optimization)

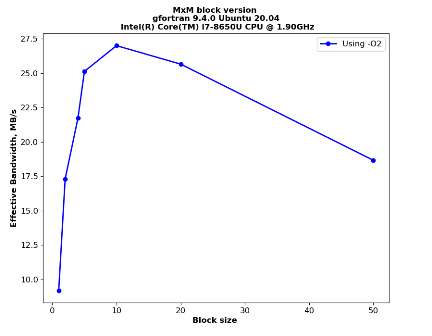
```
for ii in range(0, n, nb):
    for jj in range(0, n, nb):
        for kk in range(0, n, nb):
            for i in range(ii, min(ii+nb, n)):
                for j in range(jj, min(jj+nb, n)):
                    for k in range(kk, min(kk+nb, n)):
                        c[i][j] +=a[i][k] *b[k][j]
```



BLOCK MATRIX MULTIPLICATION

mxm block version: CPU time & Bandwidth





SUMMARY

- Access contiguous, stride-one memory addresses
- Emphasize cache reuse
- Use data structures that improve locality
- Minimize communication across different memory levels
- Use parallelism to improve locality

OUTLINE

- About Python
- Python is slow !
- Profiling a Python code

ABOUT PYTHON

- Python was created by Guido van Rossum in 1991 (last version 3.11 - 24/10/2022)
- Python is simple
- Python is fully featured
- Python is readable
- Python is extensible
- Python is ubiquitous, portable, and free
- Python has many third party libraries, tools, and a large community

ABOUT PYTHON

- Python was created by Guido van Rossum in 1991 (last version 3.11 - 24/10/2022)
- Python is simple
- Python is fully featured
- Python is readable
- Python is extensible
- Python is ubiquitous, portable, and free
- Python has many third party libraries, tools, and a large community

➠ But Python is slow!!

ABOUT PYTHON

- Python was created by Guido van Rossum in 1991 (last version 3.11 - 24/10/2022)
- Python is simple
- Python is fully featured
- Python is readable
- Python is extensible
- Python is ubiquitous, portable, and free
- Python has many third party libraries, tools, and a large community

➠ When does it really matter?

When does it matter?

- Is my code fast?
- How many CPUh?
- Problems on the system?
- How much effort is it to make it run faster?

PROFILING A PYTHON CODE: WHY?

- Code bottlenecks
- Premature optimization is the root of all evil D. Knuth
- First make it work. Then make it right. Then make it fast. K. Beck
- How?

PROFILING A PYTHON CODE: PROFILERS

- Deterministic and statistical profiling
  - the profiler will be monitoring all the events
  - it will sample after time intervals to collect that information
- The level at which resources are measured; module, function or line level
- Profile viewers

PROFILING A PYTHON CODE: TOOLS

- Inbuilt timing modules
- profile and cProfile
- pstats
- line_profiler
- snakeviz

PROFILING A PYTHON CODE: USE CASE

```python
def linspace(start, stop, n):
  step =float(stop -start) / (n -1)
  return [start +i *step for i in range(n)]

def mandel(c, maxiter):
  z =c
  for n in range(maxiter):
    if abs(z) >2:
      return n
    z =z*z +c
  return n

def mandel_set(xmin=-2.0, xmax=0.5, ymin=-1.25, ymax=1.25,
               width=1000, height=1000, maxiter=80):
  r =linspace(xmin, xmax, width)
  i =linspace(ymin, ymax, height)
  n =[[0]*width for _ in range(height)]
  for x in range(width):
    for y in range(height):
      n[y][x] =mandel(complex(r[x], i[y]), maxiter)
  return n
```

PROFILING A PYTHON CODE: TIMEIT

The very naive way

```python
import timeit

start_time =timeit.default_timer()
mandel_set()
end_time =timeit.default_timer()
# Time taken in seconds
elapsed_time =end_time -start_time

print('> Elapsed time', elapsed_time)
```

or using the magic method `timeit`

```
[In] %timeit mandel_set()
[Out] 3.01 s +/- 84.6 ms per loop (mean +/- std. dev. of 7 runs, 1 loop each)
```

PROFILING A PYTHON CODE: PRUN

```
[In] %prun -s cumulative mandel_set()
```

which is, in console mode, equivalent to

```
python -m cProfile -s cumulative mandel.py
```

```
         25214601 function calls in 5.151 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    5.151    5.151 {built-in method builtins.exec}
        1    0.002    0.002    5.151    5.151 <string>:1(<module>)
        1    0.291    0.291    5.149    5.149 <ipython-input-4-9421bc2016cb>:13(mandel_set)
  1000000    3.461    0.000    4.849    0.000 <ipython-input-4-9421bc2016cb>:5(mandel)
 24214592    1.388    0.000    1.388    0.000 {built-in method builtins.abs}
        1    0.008    0.008    0.008    0.008 <ipython-input-4-9421bc2016cb>:17(<listcomp>)
        2    0.000    0.000    0.000    0.000 <ipython-input-4-9421bc2016cb>:1(linspace)
        2    0.000    0.000    0.000    0.000 <ipython-input-4-9421bc2016cb>:3(<listcomp>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

PROFILING A PYTHON CODE: LINE LEVEL

- Use the `line_profiler` package

```
[In] %load_ext line_profiler
[In] %lprun -f mandel mandel_set()
```

```
Timer unit: 1e-06 s
Total time: 12.4456 s
File: <ipython-input-2-9421bc2016cb>
Function: mandel at line 5
#Line       Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     5                                           def mandel(c, maxiter):
     6    1000000     250304.0      0.3      1.1      z = c
     7   24463110    6337732.0      0.3     27.7      for n in range(maxiter):
     8   24214592    8327289.0      0.3     36.5          if abs(z) > 2:
     9     751482     201108.0      0.3      0.9              return n
    10   23463110    7658255.0      0.3     33.5          z = z*z + c
    11     248518      65444.0      0.3      0.3      return n
```

PROFILING A PYTHON CODE: LINE LEVEL

This can be done in console mode as well

```
@profile
def mandel(c, maxiter):
  z = c
  for n in range(maxiter):
    if abs(z) > 2:
      return n
    z = z*z + c
  return n
```

Then on the command line

```
kernprof -l -v mandel.py
```

Then

```
python3 -m line_profiler mandel.py.lprof
```

PROFILING A PYTHON CODE: MEMORY

- Use the `memory_profiler` package

```
[In] %load_ext memory_profiler
[In] %mprun -f mandel mandel_set()
```

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     8    118.2 MiB -39057.7 MiB     1000000   def mandel(c, maxiter):
     9    118.2 MiB -39175.5 MiB     1000000     z = c
    10    118.2 MiB -293081.8 MiB    24463110     for n in range(maxiter):
    11    118.2 MiB -292425.7 MiB    24214592       if abs(z) > 2:
    12    118.2 MiB -38519.6 MiB      751482         return n
    13    118.2 MiB -253906.1 MiB    23463110       z = z*z + c
    14    118.2 MiB   -656.4 MiB      248518     return n
```

PROFILING A PYTHON CODE: MEMORY

- Use the `memory_profiler` package

```
@profile
def mandel(c, maxiter):
    z = c
    for n in range(maxiter):
        if abs(z) > 2:
            return n
        z = z*z + c
    return n
```

Then on the command line

```
mprof run mandel.py
```

Then

```
mprof plot
```

Or

```
python3 -m memory_profiler mandel.py
```

OUTLINE

- Accelerate a Python code
  - Using Numpy
  - Using Cython
  - Using Numba
  - Using Pyccel
- Some Benchmarks

ACCELERATE A PYTHON CODE: NUMPY

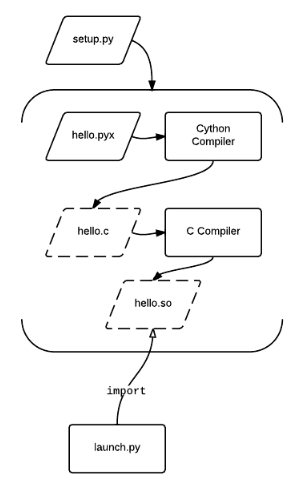
- Library for scientific computing in Python,
- High-performance multidimensional array object,
- Integrates C, C++, and Fortran codes in Python,
- Uses multithreading.

ACCELERATE A PYTHON CODE: NUMPY VS LISTS

```python
import numpy, time

size =1000000

print("Concatenation: ")
list1 =[i for i in range(size)]; list2 =[i for i in range(size)]

array1 =numpy.arange(size); array2 =numpy.arange(size)

# List
initialTime =time.time()
list1 =list1 +list2
# calculating execution time
print("Time taken by Lists :", (time.time() -initialTime), "seconds")

# Numpy array
initialTime =time.time()
array =numpy.concatenate((array1, array2), axis =0)
# calculating execution time
print("Time taken by NumPy Arrays :", (time.time() -initialTime), "seconds")
```

```
Concatenation:
Time taken by Lists : 0.021048307418823242 seconds
Time taken by NumPy Arrays : 0.009451150894165039 seconds
```

ACCELERATE A PYTHON CODE: CYTHON



- Cython is an optimizing static compiler for:
  - Python programming language
  - Cython programming language (based on Pyrex)
- Cython gives you the combined power of Python.

ACCELERATE A PYTHON CODE: CYTHON

- Python

```
def mandelbrot(m, size, iterations):
    for i in range(size):
        for j in range(size):
            c =-2 +3./size*j +1j*(1.5-3./size*i)
            z =0
            for n in range(iterations):
                if np.abs(z) <=10:
                    z =z*z +c; m[i, j] =n
                else:
                    break
```

ACCELERATE A PYTHON CODE: CYTHON

- Cython

```
def mandelbrot_cython(int[:,::1] m,int size, int iterations):
    cdef int i, j, n
    cdef complex z, c
    for i in range(size):
        for j in range(size):
            c =-2 +3./size*j +1j*(1.5-3./size*i)
            z =0
            for n in range(iterations):
                if z.real**2 +z.imag**2 <=100:
                    z =z*z +c; m[i, j] =n
                else:
                    break
```

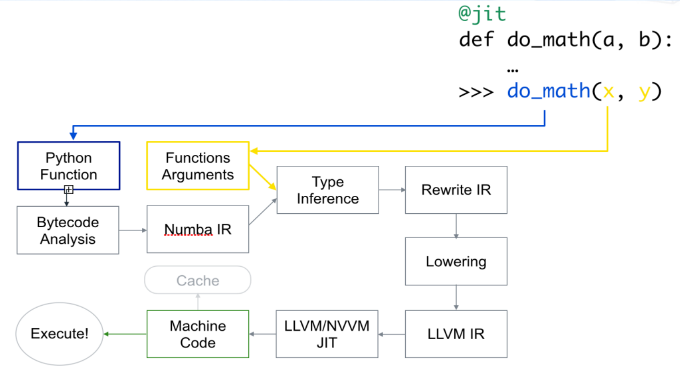
ACCELERATE A PYTHON CODE: CYTHON

- Execution time

```
%%timeit -n1 -r1
m = np.zeros(s, dtype=np.int32)
mandelbrot(m, size, iterations)
>> 12.2 s +/- 0 ns per loop (mean +/- std. dev. of 1 run, 1 loop each)


%%timeit -n1 -r1
m = np.zeros(s, dtype=np.int32)
mandelbrot_cython(m, size, iterations)
>> 29.8 ms +/- 0 ns per loop (mean +/- std. dev. of 1 run, 1 loop each)
```

ACCELERATE A PYTHON CODE: NUMBA

- Open source Just-In-Time compiler for python functions.
- Uses the LLVM library as the compiler backend.



ACCELERATE A PYTHON CODE: NUMBA

- Python

```
import numpy as np

def do_sum():
    acc =0.
    for i in range(10000000) :
        acc +=np.sqrt(i)
    return acc
```

- Numba

```
from numba import njit

@njit
def do_sum_numba():
    acc =0.
    for i in range(10000000) :
        acc +=np.sqrt(i)
    return acc
```

```
Time for Pure Python Function: 7.724030017852783
Time for Numba Function: 0.015453100204467773
```

ACCELERATE A PYTHON CODE: PYCCEL

- Pyccel is a static compiler for Python 3, using Fortran or C as a backend language.
- Python function:

```
import numpy as np

def do_sum_pyccel():
    acc =0.
    for i in range(10000000) :
        acc +=np.sqrt(i)
    return acc
```

ACCELERATE A PYTHON CODE: PYCCEL (F90)

- Compilation using fortran:

```
pyccel --language=fortran pyccel_example.py
```

```fortran
module pyccel_example
use, intrinsic :: ISO_C_Binding, only : i64 => C_INT64_T , f64 => C_DOUBLE
    implicit none
    contains
    !........................................
    function do_sum_pyccel() result(acc)

        implicit none
        real(f64) :: acc
        integer(i64) :: i
        acc = 0.0_f64
        do i = 0_i64, 9999999_i64, 1_i64
            acc = acc + sqrt(Real(i, f64))
        end do
        return
    end function do_sum_pyccel
!........................................
end module pyccel_example
```

```
Time for Pure Python Function: 7.400242328643799
Time for Pyccel Function: 0.01545262336730957
```

ACCELERATE A PYTHON CODE: PYCCEL (C)

- Compilation using c:

```
pyccel --language=c pyccel_example.py
```

```c
#include "pyccel_example.h"
#include <stdlib.h>
#include <math.h>
#include <stdint.h>
/*........................................*/
double do_sum_pyccel(void)
{
    int64_t i;
    double acc;
    acc = 0.0;
    for (i = 0; i < 10000000; i += 1)
    {
        acc += sqrt((double)(i));
    }
    return acc;
}
/*........................................*/
```

SOME BENCHMARKS

Rosen-Der

Tool	Python	Cython	Numba	Pythran	Pyccel-gcc	Pyccel-intel
Timing (μs)	229.85	2.06	4.73	2.07	0.98	0.64
Speedup	—	$\times$ 111.43	$\times$ 48.57	$\times$ 110.98	$\times$ 232.94	$\times$ 353.94

Black-Scholes

Tool	Python	Cython	Numba	Pythran	Pyccel-gcc	Pyccel-intel
Timing (μs)	180.44	309.67	3.0	1.1	1.04	$6.56\ 10^{-2}$
Speedup	—	$\times$ 0.58	$\times$ 60.06	$\times$ 163.8	$\times$ 172.35	$\times$ 2748.71

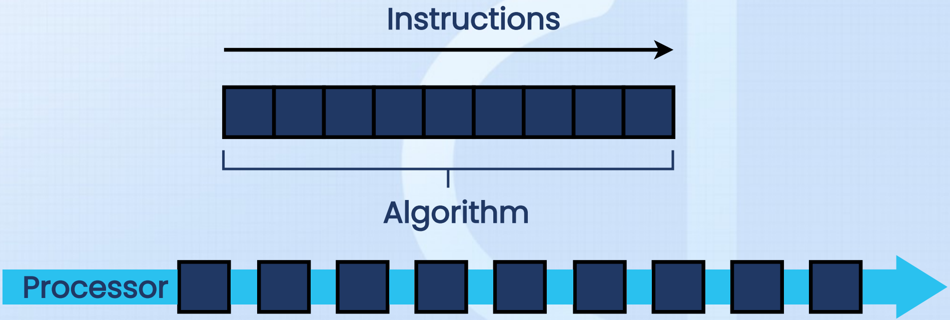
Laplace

Tool	Python	Cython	Numba	Pythran	Pyccel-gcc	Pyccel-intel
Timing (μs)	57.71	7.98	$6.46\ 10^{-2}$	$6.28\ 10^{-2}$	$8.02\ 10^{-2}$	$2.81\ 10^{-2}$
Speedup	—	$\times$ 7.22	$\times$ 892.02	$\times$ 918.56	$\times$ 719.32	$\times$ 2048.65

OUTLINE

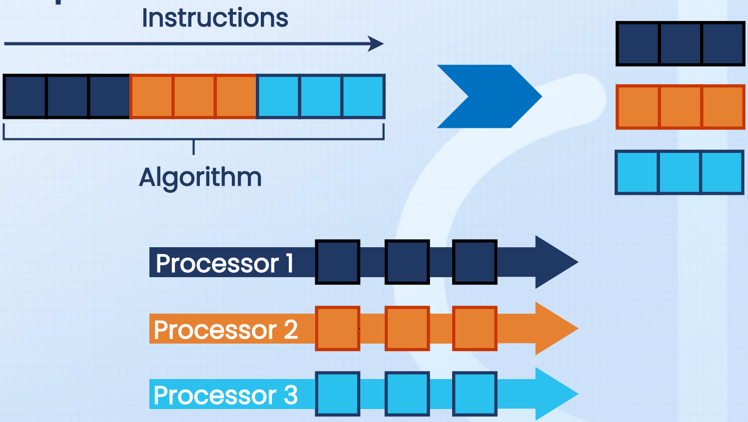
- Parallel Programming
- Parallel strategies
- Parallel infrastructures

Serial problem



PARALLEL PROGRAMMING

Parallel problem



PARALLEL STRATEGIES

Setting

PARALLEL STRATEGIES

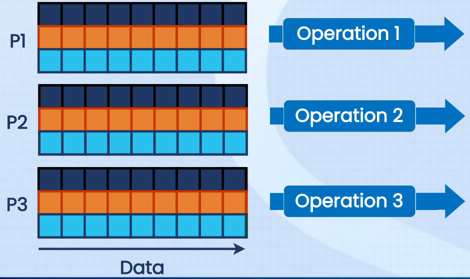
Setting

Data parallelism

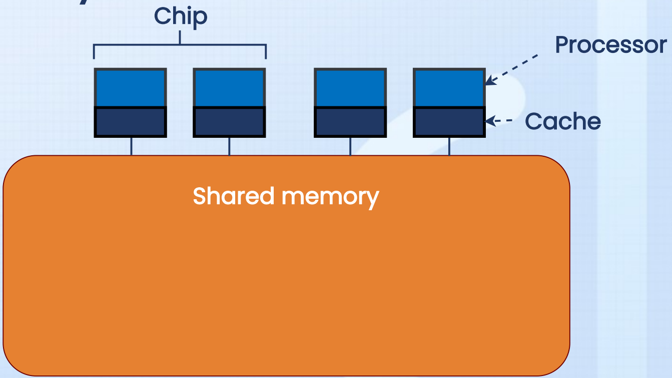
PARALLEL STRATEGIES

Setting

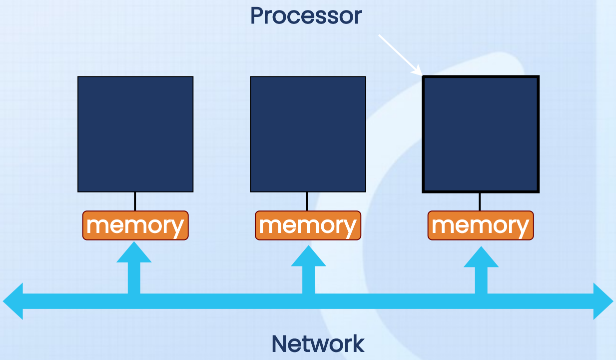
Task parallelism



Shared memory



Distributed memory



SUMMARY

- Advantages
  - Parallel computing saves time
  - Solve Larger Problems
  - Data storage
- Walls
  - Writing parallel codes is time-consuming
  - Power consumption
  - CPUs, GPUs, FPGAs

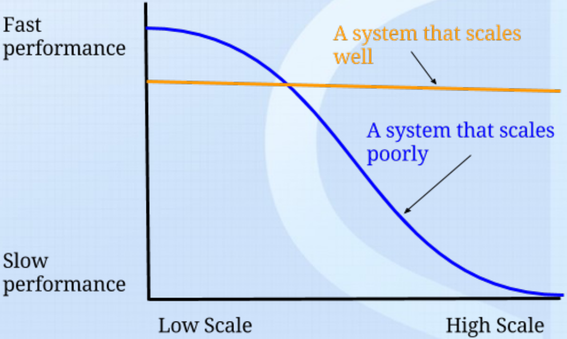
OUTLINE

- Performance and scalability
- Classes of performance metrics
- Execution time & Total Parallel Overhead
- Speedup & Efficiency
- Amdahl’s & Gustafson’s Laws

PERFORMANCE AND SCALABILITY

Design of parallel applications:

- Performance
- Scalability



CLASSES OF PERFORMANCE METRICS

Distinct classes of performance metrics:

- Performance metrics for processors/cores
- Performance metrics for parallel applications
  - Practical metrics
  - Theoretical metrics

EXECUTION TIME

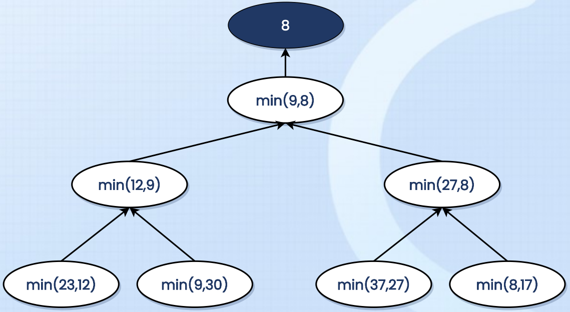
Finding minimum element among [23,12,9,30,37,27,8,17].

- Execution time:
  - Serial Time T_S $(\theta(n))$

EXECUTION TIME

Finding minimum element among [23,12,9,30,37,27,8,17].

- Execution time:
  - Serial Time T_S $(\theta(n))$
  - Parallel Time T_P $(\theta(\log n))$



TOTAL PARALLEL OVERHEAD

Finding minimum element among [23,12,9,30,37,27,8,17].

- Execution time:
  - Serial Time T_s $(\theta(n))$
  - Parallel Time T_p $(\theta(\log n))$

Total Parallel Overhead

- With p processes:
  - Total time = pT_p
  - Overhead = $pT_p - T_s$

SPEEDUP

Speedup is a measure of performance.

$$S_p = \frac{T_s}{T_p}$$

- Example 1: Find out the minimum element in array

$$S_p = \frac{\theta(n)}{\theta(\log n)} = \theta\left(\frac{n}{\log n}\right)$$

- Example 2: Solve 1D transport equation

	1 CPUs	2 CPUs	4 CPUs	8 CPUs	16 CPUs
T(p)	1000	520	280	160	100
S(p)	1	1.92	3.57	6.25	10.00

EFFICIENCY

Efficiency is a measure of the usage of the computational capacity.

$$E_p = \frac{S_p}{p} = \frac{T_s}{p \times T_p}$$

- Example 1: Find out the minimum element in array

$$E_p = \frac{\theta\left(\frac{n}{\log n}\right)}{p} \quad (\text{if } p = n) => E_p = \frac{\theta\left(\frac{n}{\log n}\right)}{n} = \theta\left(\frac{1}{\log n}\right)$$

- Example 2: Solve 1D transport equation

	1 CPUs	2 CPUs	4 CPUs	8 CPUs	16 CPUs
S(p)	1	1.92	3.57	6.25	10.00
E(p)	1	0.96	0.89	0.78	0.63

AMDAHL'S LAW: STRONG SCALING

- Serial part: $0 \leq f \leq 1$

$$S_p = \frac{1}{f + \frac{1-f}{p}}$$

- Example: If f = 10%, what is the max. speedup achievable using 8 processors?

  - Solution:

$$S_p = \frac{1}{0.1 + \frac{1-0.1}{8}} \approx 4.7$$

  - Limit:

$$\lim_{p \to \infty} = \frac{1}{0.1 + \frac{1-0.1}{p}} = 10$$

GUSTAFSON'S LAW: WEAK SCALING

- Parallel part: $0 \le f' \le 1$

$$S_p = (1 - f') + f' \times p = 1 + (p - 1) \times f'$$

- Example: $f' = 90\%$, what is the scaled speedup using 64 processors?
- Solution:

$$S_p = 1 + (p - 1) \times f' = 1 + (64 - 1) \times 0.90 = 57.70$$

SUMMARY

- Scalable application
  - Strong scaling + Weak scaling

		1 CPUs	2 CPUs	4 CPUs	8 CPUs	16 CPUs
Efficiency	n = 10 000	1	0.81	0.53	0.28	0.16
	n = 20 000	1	0.94	0.80	0.59	0.42
	n = 40 000	1	0.96	0.89	0.79	0.58

- Superlinear Speedup

$$\frac{T_s}{T_p} \geq p$$