High-Level Parallel Programming

1. High-Level Parallel Programming
   - models and tools

2. Multiprocessing in Python
   - scripting in computational science
   - the multiprocessing module
   - numerical integration with multiple processes

3. Multithreading with Julia
   - a fresh approach to numerical computing
   - parallel matrix matrix multiplication
   - parallel numerical integration

MCS 572 Lecture 3
Introduction to Supercomputing
Jan Verschelde, 13 January 2023
High Level Parallel Processing

1. High-Level Parallel Programming
   - models and tools

2. Multiprocessing in Python
   - scripting in computational science
   - the multiprocessing module
   - numerical integration with multiple processes

3. Multithreading with Julia
   - a fresh approach to numerical computing
   - parallel matrix matrix multiplication
   - parallel numerical integration
What is high-level parallel programming?

Some characteristics:

- familiar: no new language needed,
- interactive: quick feedback,
- personal: no supercomputer.

Rapid prototyping can decide if parallelism is feasible for a particular computation in an application.
The 15th international symposium on High-Level Parallel Programming, HLPP 2022, was held in Porto, Portugal, July 7-8, 2022.

Some of the topics include:

- high-level programming and performance models,
- software synthesis, automatic code generation,
- applications using high-level languages and tools,
- formal models of verification.

While “high-level” also covers abstract and formal, there is a need for practical software and tools, so the “high-level” is not the opposite of technical.
High Level Parallel Processing

1. High-Level Parallel Programming
   - models and tools

2. Multiprocessing in Python
   - scripting in computational science
   - the multiprocessing module
   - numerical integration with multiple processes

3. Multithreading with Julia
   - a fresh approach to numerical computing
   - parallel matrix matrix multiplication
   - parallel numerical integration
Advantages of the scripting language Python:
- educational, good for novice programmers;

SageMath, a free open source mathematics software system, uses Python to interface many open source software packages.

Our example: \[ \int_0^1 \sqrt{1 - x^2} \, dx = \frac{\pi}{4}. \]

We will use the Simpson rule (available in \texttt{SciPy}) as a relatively computational intensive example.
interactive computing

In a Powershell window:

Python 3.10.1 (tags/v3.10.1:2cd268a, Dec 6 2021, 19:10:37) MSC v.1929 64 bit (AMD64) on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from scipy.integrate import simps
>>> from numpy import linspace
>>> from numpy.lib.scimath import sqrt
>>> f = lambda x: sqrt(1-x**2)
>>> r = linspace(0,1,1000)
>>> y = f(r)
>>> I = simps(y,r)
>>> 4*I
3.1415703366671104
>>>
The script `simpson4pi.py`

```python
from scipy.integrate import simps
from numpy import linspace, pi
from numpy.lib.scimath import sqrt

f = lambda x: sqrt(1-x**2)

for k in range(2,9):
    x = linspace(0,1,10**k);
    y = f(x);
    I = 4*simps(y,x)
    print('10^%d' % k, \
          '%.16e' % I, \
          '%.2e' % abs(I - pi))
```

Introduction to Supercomputing (MCS 572)
running the script simpson4pi.py

The output on screen:

10^2 3.1408763613344828e+00 7.16e-04
10^3 3.1415703366671104e+00 2.23e-05
10^4 3.1415919488981889e+00 7.05e-07
10^5 3.1415926313087348e+00 2.23e-08
10^6 3.1415926528852145e+00 7.05e-10
10^7 3.1415926535675127e+00 2.23e-11
10^8 3.1415926535890946e+00 6.99e-13

Getting the execution times:

- on Linux and Mac: time python simpson4pi.py
- on Windows: Measure-Command {python simpson4pi.py}
timing on Windows

Measure-Command {python simpson4pi.py}

Days : 0
Hours : 0
Minutes : 0
Seconds : 5
Milliseconds : 785
Ticks : 57852572
TotalDays : 6.69589953703704E-05
TotalHours : 0.00160701588888889
TotalMinutes : 0.0964209533333333
TotalSeconds : 5.7852572
TotalMilliseconds : 5785.2572

Intel i9-9880H CPU at 2.30Ghz, 8 cores, 16 logical processors, 32.0 GB internal memory, Microsoft Windows 11.
Isolating the last run:

```python
from scipy.integrate import simps
from numpy import linspace, pi
from numpy.lib.scimath import sqrt
f = lambda x: sqrt(1-x**2)
r = linspace(0, 1, 10**8)
y = f(r)
I = 4*simps(y, r)
print('%.16e' % I, '%.2e' % abs(I - pi))
```

Measure-Command {python simpson4pi1.py} reports 5.200 seconds.

*Can we improve this?*
High Level Parallel Processing

1. High-Level Parallel Programming
   - models and tools

2. Multiprocessing in Python
   - scripting in computational science
   - the multiprocessing module
   - numerical integration with multiple processes

3. Multithreading with Julia
   - a fresh approach to numerical computing
   - parallel matrix matrix multiplication
   - parallel numerical integration
With multiprocessing we run multiple processes simultaneously.

- Each process acts as a separate program.
- The `multiprocessing` module of Python enables
  - the launching of processes,
  - interprocess communication.

The multithreading in Python supports concurrency — think of the polite dinner conversation — not true parallelism, because of the interpreter lock.
from multiprocessing import Process
import os
from time import sleep

def say_hello(name, t):
    
    """
    Process with name says hello.
    """

    print('hello from', name)
    print('parent process :', os.getppid())
    print('process id :', os.getpid())
    print(name, 'sleeps', t, 'seconds')
sleep(t)
    print(name, 'wakes up')
creating the processes

The script continues:

```python
pA = Process(target=say_hello, args = (‘A’,2,))
pB = Process(target=say_hello, args = (‘B’,1,))
pA.start(); pB.start()
print(‘waiting for processes to wake up...’)  
pA.join(); pB.join()
print(‘processes are done’)
```
running the script

The output of `python multiprocessing.py` is

waiting for processes to wake up...
hello from A
parent process : 737
process id : 738
A sleeps 2 seconds
hello from B
parent process : 737
process id : 739
B sleeps 1 seconds
B wakes up
A wakes up
processes are done
High Level Parallel Processing

1. High-Level Parallel Programming
   - models and tools

2. Multiprocessing in Python
   - scripting in computational science
   - the multiprocessing module
   - numerical integration with multiple processes

3. Multithreading with Julia
   - a fresh approach to numerical computing
   - parallel matrix matrix multiplication
   - parallel numerical integration
We apply domain partitioning:

\[
\int_a^b f(x)\,dx = \sum_{i=0}^{p-1} \int_{a+i\Delta}^{a+(i+1)\Delta} f(x)\,dx
\]

where

\[
\Delta = \frac{b-a}{p}, \quad p \geq 1.
\]
from multiprocessing import Process, Queue
from scipy.integrate import simps
from numpy import linspace, pi
from numpy.lib.scimath import sqrt

def call_simpson(fun, a, b, n, q):
    """
    Calls Simpson rule to integrate fun over [a, b] using n intervals.
    Adds the result to the queue q.
    """
    x = linspace(a, b, n)
    y = fun(x)
    I = simps(y, x)
    q.put(I)
the main program

def main():
    
    The number of processes is given at the command line.
    
    from sys import argv
    if len(argv) < 2:
        print('Enter the number of processes')
        print('at the command line.')
        return
    npr = int(argv[1])

We want to run the script as

```
python simpson4pi2.py 4
```

to time the running of the script with 4 processes.
defining processes and queues

crc = lambda x: sqrt(1-x**2)
nbr = 10**8
nbrsam = nbr//npr
intlen = 1.0/npr
queues = [Queue() for _ in range(npr)]
procs = []
(left, right) = (0, intlen)
for k in range(1, npr+1):
    procs.append(Process(target=call_simpson, 
                        args = (crc, left, right, nbrsam, queues[k-1])))
    (left, right) = (right, right+intlen)
starting processes and collecting results

```python
for process in procs:
    process.start()
for process in procs:
    process.join()
app = 4*sum([q.get() for q in queues])
print('%.16e' % app, '%.2e' % abs(app - pi))
```
checking for speedup

Measure-Command \{python3 simpson4pi1.py\} resulted in 5.200 seconds.

Now we run

Measure-Command \{python simpson4pi2.py 2\}
Measure-Command \{python simpson4pi2.py 4\}
Measure-Command \{python simpson4pi2.py 8\}

to find 3.985, 3.491, and 3.518 respectively.

Computing the speedups:

1. $5.200/3.985 \approx 1.30$ with $p = 2$,
2. $5.200/3.985 \approx 1.49$ with $p = 4$,
3. $5.200/3.985 \approx 1.48$ with $p = 8$. 
running times and speedups on a fast workstation

Times in seconds obtained as `time python3 simpson4pi2.py p` for \( p = 2, 4, 8, 16, \) and 32, on two 22-core Intel Xeon E5-2699v4 Broadwell at 2.20GHz, with 256GB of internal memory at 2400MHz.

For \( p = 1 \), `time python3 simpson4pi1.py` was used.

<table>
<thead>
<tr>
<th>( p )</th>
<th>real</th>
<th>user</th>
<th>sys</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.586</td>
<td>7.766</td>
<td>14.135</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3.513</td>
<td>7.377</td>
<td>12.589</td>
<td>1.87</td>
</tr>
<tr>
<td>4</td>
<td>2.011</td>
<td>7.154</td>
<td>11.701</td>
<td>3.27</td>
</tr>
<tr>
<td>8</td>
<td>1.275</td>
<td>8.043</td>
<td>12.920</td>
<td>5.17</td>
</tr>
<tr>
<td>16</td>
<td>0.953</td>
<td>10.095</td>
<td>12.893</td>
<td>6.91</td>
</tr>
<tr>
<td>32</td>
<td>0.904</td>
<td>14.154</td>
<td>13.915</td>
<td>7.29</td>
</tr>
</tbody>
</table>

Speedups are computed as \( \frac{\text{real time with } p = 1}{\text{real time with } p \text{ tasks}} \).
High Level Parallel Processing

1. High-Level Parallel Programming
   - models and tools

2. Multiprocessing in Python
   - scripting in computational science
   - the multiprocessing module
   - numerical integration with multiple processes

3. Multithreading with Julia
   - a fresh approach to numerical computing
   - parallel matrix matrix multiplication
   - parallel numerical integration
What makes Julia great?

- When coded well, it is very fast
- Great ability to mix loop based & matrix/vector operations
- Clear, concise code that can easily be changed

-√ Java
-Δ Python (Cython, etc)
-Δ R (vectorized)

-Δ Java (not really)
-√ Python
-Δ R (only vectorized)

-Δ Java (not concise)
-√ Python
-Δ R (only R code, not C or C++)
High Level Parallel Processing

1. High-Level Parallel Programming
   - models and tools

2. Multiprocessing in Python
   - scripting in computational science
   - the multiprocessing module
   - numerical integration with multiple processes

3. Multithreading with Julia
   - a fresh approach to numerical computing
   - parallel matrix matrix multiplication
   - parallel numerical integration
We apply multithreading in a Jupyter notebook, in a kernel installed with the environment variable set to 16 threads.

```julia
julia> using IJulia
julia> installkernel("Julia (16 threads)",
    env = Dict("JULIA_NUM_THREADS"=>"16"))
```

The matrix-matrix multiplication is executed by `mul!()` of BLAS, where BLAS stands for the Basic Linear Algebra Subroutines.

Two issues we must consider:

1. Choose the size of the matrices large enough.
2. The time should not include the compilation time.
Parallel Matrix Matrix Multiplication

The instructions in code cells of a Jupyter notebook:

```julia
using LinearAlgebra
n = 8000
A = rand(n, n);
B = rand(n, n);
C = rand(n, n);
BLAS.set_num_threads(2)
@time mul!(C, A, B)
```

10.722 seconds (2.87 M allocations, 5.13% compilation time)

Redo, the second time: 10.359 seconds.

```julia
BLAS.set_num_threads(4)
@time mul!(C, A, B)
```

6.080 seconds.
High Level Parallel Processing

1. High-Level Parallel Programming
   - models and tools

2. Multiprocessing in Python
   - scripting in computational science
   - the multiprocessing module
   - numerical integration with multiple processes

3. Multithreading with Julia
   - a fresh approach to numerical computing
   - parallel matrix matrix multiplication
   - parallel numerical integration
Numerical Integration

We can estimate $\pi$, via the area of the unit disk:

$$\int_0^1 \sqrt{1 - x^2} \, dx = \frac{\pi}{4}$$

1. Generate random uniformly distributed points with coordinates $(x, y) \in [0, +1] \times [0, +1]$.

2. We count a success when $x^2 + y^2 \leq 1$.

By the law of large numbers, the average of the observed successes converges to the expected value or mean, as the number of experiments increases.
A dedicated random number generator is applied:

\[
\text{myrand}(x::\text{Int64}) = (1103515245x + 12345) \mod 2^{31}
\]

The specification of the function is below:

```
    function estimatepi(n)
        Runs a simple Monte Carlo method to estimate pi with n samples.
    end
```

the function \texttt{estimatepi}(n)

\begin{verbatim}
function estimatepi(n)
    r = threadid()
    count = 0
    for i=1:n
        r = myrand(r)
        x = r/2^31
        r = myrand(r)
        y = r/2^31
        count += (x^2 + y^2) <= 1
    end
    return 4*count/n
end
\end{verbatim}
running in a notebook with 16 threads

Observe the parallel for loop:

```plaintext
nt = nthreads()
estimates = zeros(nt)
import Statistics
timestart = time()

@threads for i=1:nt
    estimates[i] = estimatepi(10_000_000_000/nt)
end

estpi = Statistics.mean(estimates)
elapsed16 = time() - timestart

5.387
```
running on many threads on a fast workstation

Running version 1.4.0-DEV.364 (2019-10-22) on two 22-core 2.2 GHz Intel Xeon E5-2699 processors in a CentOS Linux workstation with 256 GB RAM.

<table>
<thead>
<tr>
<th>$p$</th>
<th>wall clock time</th>
<th>elapsed time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1m 2.313s</td>
<td>62.060s</td>
</tr>
<tr>
<td>2</td>
<td>32.722s</td>
<td>32.418s</td>
</tr>
<tr>
<td>3</td>
<td>22.471s</td>
<td>22.190s</td>
</tr>
<tr>
<td>4</td>
<td>17.343s</td>
<td>17.042s</td>
</tr>
<tr>
<td>5</td>
<td>14.170s</td>
<td>13.896s</td>
</tr>
<tr>
<td>6</td>
<td>12.300s</td>
<td>11.997s</td>
</tr>
<tr>
<td>7</td>
<td>10.702s</td>
<td>10.442s</td>
</tr>
</tbody>
</table>
Supercomputing is for everyone as most modern software provides options and tools to run on parallel machines.

- Python is a good prototyping language to define and try parallel algorithms on multicore workstations.
- Julia is a new programming language for scientific computing designed for performance.


This lecture had its focus on multiprocessing and multithreading; Python and Julia support distributed memory parallel computing.
A Monte Carlo method to estimate $\pi/4$ generates random tuples $(x, y)$, with $x$ and $y$ uniformly distributed in $[0, 1]$. The ratio of the number of tuples inside the unit circle over the total number of samples approximates $\pi/4$.

```python
>>> from random import uniform as u
>>> X = [u(0,1) for i in xrange(1000)]
>>> Y = [u(0,1) for i in xrange(1000)]
>>> Z = zip(X,Y)
>>> F = filter(lambda t: t[0]**2 + t[1]**2 <= 1, Z)
>>> len(F)/250.0
3.1440000000000001
```

Use the multiprocessing module to write a parallel version, letting processes take samples independently. Compute the speedup.

Develop a parallel Julia version for the `simpson4pi` code.