UNIVERSIDAD POLITÉCNICA DE YUCATÁN

BIS Universities

PARALLEL PROGRAMMING WITH PYTHON

HIGH PERFORMANCE COMPUTING

April 19th, 2024

Made by

Karla Delgado Avendaño

Parallel Programming with Python

Karla Delgado
Data Engineering
Universidad Politécnica de Yucatán
Ucú, Yucatán, México
2109050@upy.edu.mx

I. Introduction

The Riemann sum approach offers a straightforward means to estimate the value of π through the integration of the quarter-circle area. This paper examines the efficiency of this method when implemented in Python, particularly in the context of parallel computing. By employing different strategies of parallel programming, including multiprocessing and distributed computing, we aim to optimize the computational performance on multicore systems, elaborating on the mathematical foundation of the approximation and also the implementation of specififc librabries and methods related to the structured used when coding with Python

II. Mathematical Background

The calculation of π can be approximated numerically by integrating the area of a quarter of a unit circle. This area under the curve can be calculated using the Riemann sum approach. For a unit circle described by the equation $x^2+y^2 = 1$, the quarter circle in the first quadrant is represented by the function $f(x) = \sqrt{1-x^2}$.

To compute the area under the curve of $f(x)$ from $x = 0$ to $x = 1$, we discretize the interval into N subintervals of equal width $\Delta x = \frac{1}{N}$. The approximation of the area under $f(x)$, and hence the approximation of $\frac{\pi}{4}$, is given by the sum:

$$\frac{\pi}{4} \approx \sum_{i=0}^{N-1} f(x_i)\Delta x \tag{1}$$

where $x_i = i\Delta x$ represents the i-th subinterval's right endpoint. The summation effectively adds up the areas of rectangles with height $f(x_i)$ and width Δx, providing an estimate of the quarter-circle's area.

As N approaches infinity, the Riemann sum converges to the actual area under the curve, yielding the value of $\frac{\pi}{4}$ with increasing accuracy.

III. Solution Description

To properly view and analyze the different ways in which parallel programming works , we implement three Python programs to solve the Riemman Sum problem:

1) A pure Python version.
2) A version using the `multiprocessing` module for parallel computing.
3) A version using `mpi4py` for distributed parallel computing.

A. Pure Python Implementation

This section talks about a Python function named `compute_pi` that guesses the value of π. The function takes a number N which is how many pieces we break the circle into to guess its size. Inside `compute_pi`, we work out how wide each piece should be, which we call `delta_x`, by splitting 1 into N pieces. We have a starting point called `area` set to zero, where we will keep adding the size of each piece.

Then there's a loop that runs N times, once for every piece. In this loop, we calculate how tall each piece is—this is like measuring how high the curve of the circle is at that point. We do this with a math formula that involves squaring and taking the square root. Every time we loop through, we add the size of the rectangle (its height times its width) to the `area`. After we've added up all the pieces, we make the `area` four times bigger because we've only worked out a quarter of the circle so far. This bigger number is what we think π is, and we call it `pi_approx`.

Outside the function, `compute_pi` is called with $10,500,000$ as an argument to calculate π with high precision.

B. Multiprocessing Implementation

The second script made was a Python program that uses several processors at the same time to estimate the value of π. This method is faster because it does many calculations at once. First, we get tools from a Python package called 'multiprocessing' and Numpy (just like in the previous code0. The `f(x)` function is defined to do the math for our curve, the top part of a circle.

Then there's `chunk_integration`, a function to add up a bunch of rectangles' areas. It takes a start point, an end point, and the width of each rectangle (`dx`). It creates a list of x-values from start to end, without including the end, and calculates the total area of the rectangles for these x-values.

The main function is `compute_pi_parallel`. It uses several processors to calculate π. It takes two numbers: how many processors to use and how many slices to divide the circle into. Inside, it works out the width of each rectangle (`dx`) by dividing 1 by the number of slices. It then decides how many slices each processor should work on (`chunk_size`) and makes a list of these slices for each processor (`chunks`). If there are any slices left over, the code adds an extra chunk for them.

Next, it sets up a pool of processors. Each processor takes a chunk of work from the list and does the calculations. The

results from all the processors are then added together and multiplied by 4 because we only calculated a quarter of the whole circle, and this gives us our estimate of π. When running the script, the function was able to get a very accurate estimate of π, using 4 processors and 10,500,000 "slices"

C. Distributed Parallel Computing with mpi4py

This Python script is also applied to estimate the value of π, but now using distributed parallel computing with the `mpi4py` library by spreading out calculations across multiple processors and then combines the results.

The MPI environment is set up using `COMM_WORLD`, with `rank` identifying each processor and `size` giving the total number of processors. The variable, `N`, is assigned the number of intervals the circle is divided into, determining the accuracy of the estimate based on the amount. Each processor calculates the width of its segment of the circle, `dx`, by dividing 1 by `N`. The script then determines the start and end points for each processor's segment of the calculation. The last processor may cover more ground if `N` is not a multiple of `size`.

Next, `numpy` is used to compute the x-values for each processor's segment and to calculate the local sum by adding up the heights at these points. All local sums are then combined into a total sum using `reduce` from `MPI`, with the root processor gathering all the sums.

If the processor is the root (rank 0), it multiplies the total sum by 4, since the calculations are for a quarter circle, to get the estimate for π and prints it.

IV. Profiling and Performance Analysis

To understand how each piece of code uses time, we kept track of how long they took to run. Let's set the scene for the results by explaining why we chose 10,500,000 as our key number. Initially, we tested the Python programs with N set to 1,000,000.

Though that's a pretty large number, the timing differences between the scripts weren't as clear as we'd hoped. So, we kept upping the number until the time differences became more obvious. We even went up to 100,000,000 to get a clear picture of time usage, which really showed the impact on speed and performance.

Running the simple Python script with that huge number was okay, but it really slowed it sown. But when we tried the same with the MPI code, the improvement was stark—it finished much quicker. However, attempting this with the multiprocessing script wasn't a success; it overwhelmed the computer to the point of freezing and crashing, which meant I had to power down the machine to sort it out.

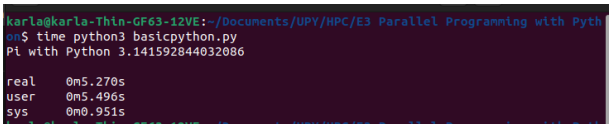
To prevent such issues, we brought the number down to 10,500,000. We felt that 10,000,000 didn't give us enough of a contrast in timing between the three pieces of code.

Now that we've settled on why 10,500,000, let's look at what the results:

TABLE I
Comparison of Computational Methods

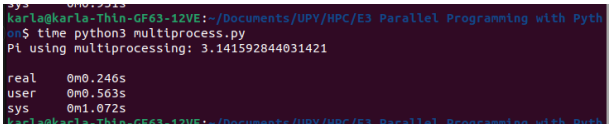
Method	Description	Time (real)	Time (sys)
Single-threaded (basicpython.py)	Computation is done in a single process without parallelization. Utilizes only one CPU core, simplest but slowest method.	5.270s	0.951s
Multiprocessing (multiprocess.py)	Uses multiple processes on the same machine, utilizing multiple CPU cores concurrently. Reduces computation time significantly, though total CPU time is higher due to summing across processes.	0.246s	1.072s
MPI (mpi4pyparallel.py)	Designed for distributed computing across multiple nodes. Suitable for scaling beyond a single machine, potentially faster than single-threaded, slower than multiprocessing but could show its potential with a bigger value of N.	0.526s	1.473s

A. Pure Python Implementation



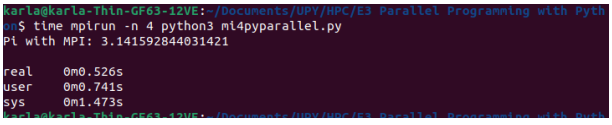
This is the slowest and most CPU-intensive for user time, indicating that the calculation is happening in user mode.

B. Multiprocessing Implementation



This results in a significantly reduced real time because multiple processors are performing the calculation concurrently. However, the user CPU time does not decrease proportionally to the real time because it sums the time on all CPUs.

C. Distributed Parallel Computing with mpi4py



The real time here is longer than the multiprocessing approach but still significantly faster than the single-threaded approach. The user CPU time is slightly less than the real time, which might indicate efficient computation.

V. Conclusion

In conclusion, the choice between single-threaded, multiprocessing, and MPI methods should be guided by the specific requirements and constraints of the computational task at hand. Multiprocessing generally offers a substantial reduction in time for single-machine applications by effectively leveraging multiple CPU cores. However, for tasks requiring even greater scale, MPI provides essential capabilities.