Universidad Politécnica de Yucatán

Gabriel Islas Peraza

Data 7ºB

High Performance Computing

Parallel Programming with Python

Teacher: Didier Omar Gamboa Angulo

Introduction

Parallel programming is essential in modern computing, as it is needed to get full power of multi-core processors and distributed computing systems to solve complex problems in a more efficient way; in this report, multiple approaches to parallel programming are compared with a normal approach in a specific problem where the value of pi is approximated.

The way it is approximated is through the use of Riemann sums; given a circle of radius 1 center at the origin, its equation is given by x^2 + y^2 = 1. Taking a look at the first quadrant of this equation, where both x and y are both greater than or equal to 0, the equation in this section can be solved for y to get y = sqrt(1 - x^2). Since this is a fourth of the unit circle, the area under this curve is then pi/4, which would be equivalent to the integral from 0 to 1 of the equation solved for y. This integral can then be expressed using Riemann sums as follows:

$$\frac{\pi}{4} \approx \sum_{i=0}^{N-1} \Delta x \, f(x_i) \text{ where } x_i = i\Delta x \text{ and } \Delta x = 1/N$$

Once this sum is calculated, multiplying by 4 will give the resulting estimation of pi. As N increases, meaning more subintervals of the integral, this approximation becomes more accurate because the rectangles fit more to the curve of the quarter circle.

As the nature of the problem shows, this is a good problem to compare parallel programming implementations, as it is a task that can clearly be divided into multiple tasks, of which their results can then be combined to create the final result. It is also very easy to scale and very simple to understand, and therefore, code.

Solutions

Normal

```
def approximate_pi(N):
      piover4 = 0
      dx = 1/N
      for i in range(N):
            xi = i * dx
            piover4 += dx * ((1 - xi**2) ** (1/2))
      return piover4 * 4
```

The normal implementation is the simple; piover4 is the variable where each rectangle of the integral will be summed up to approximate it, so a for loop from 0 to N where each value is calculated and added to said variable, then that value multiplied by 4 is returned, which is the pi estimation.

multiprocessing

```
class Process(mp.Process):
      def __init__(self, id, piover4, lock, N):
            super(Process, self).__init__()
            self.id = id
            self.piover4 = piover4
            self.lock = lock
            self.N = N

      def run(self):
            m = self.N // mp.cpu_count()
            dx = 1/self.N
            startArray = np.arange(self.id*m, (self.id+1)*m)
            xi = startArray * dx
            result = dx * ((1 - xi**2) ** (1/2))
            loc_sum = np.sum(result)
            with self.lock:
                  self.piover4.value += loc_sum
def main(N):
      piover4 = mp.Value("d", lock = True)
      piover4.value = 0
      lock = mp.Lock()

      processes = [Process(i, piover4, lock, N) for i in
```

```
range(mp.cpu_count())]
      [p.start() for p in processes]
      [p.join() for p in processes]
      result = piover4.value * 4
      print(result)
```

Using the multiprocessing module, the calculation can be divided into different instances of a custom Process class; each process has an id that also works as the definition on which process works on which part of the problem, as well as receiving the N value. All processes receive the same shared value and the same lock to avoid race conditions. The size of the problem for each process depends on the CPU count (which is 8 in the computer where the code was tested). The variables are defined as usual, except now instead of using a loop, a numpy array with the i values that the process specifically handles is created, and operations are applied on it directly for convenience. Once the numpy array receives the operations, the local sum of that numpy array is added to the shared piover4 value, taking care of race conditions by checking with the lock first. Finally, the shared value is multiplied by 4 to obtain the result.

mpi4py

```
tdef approximate_pi(N):
    # Rank
    r = MPI.COMM_WORLD.Get_rank()
    # Size
    size = MPI.COMM_WORLD.Get_size()
    # Elements in each
    m = N // size
    dx = 1/N
    startArray = np.arange(r*m, (r+1)*m)
    xi = startArray * dx
    operation = dx * ((1 - xi**2) ** (1/2))
    locsum = np.sum(operation)
    rcvBuf = np.array(0.0, "d")

    MPI.COMM_WORLD.Allreduce([locsum, MPI.DOUBLE], [rcvBuf,
MPI.DOUBLE], op = MPI.SUM)
    return rcvBuf
```

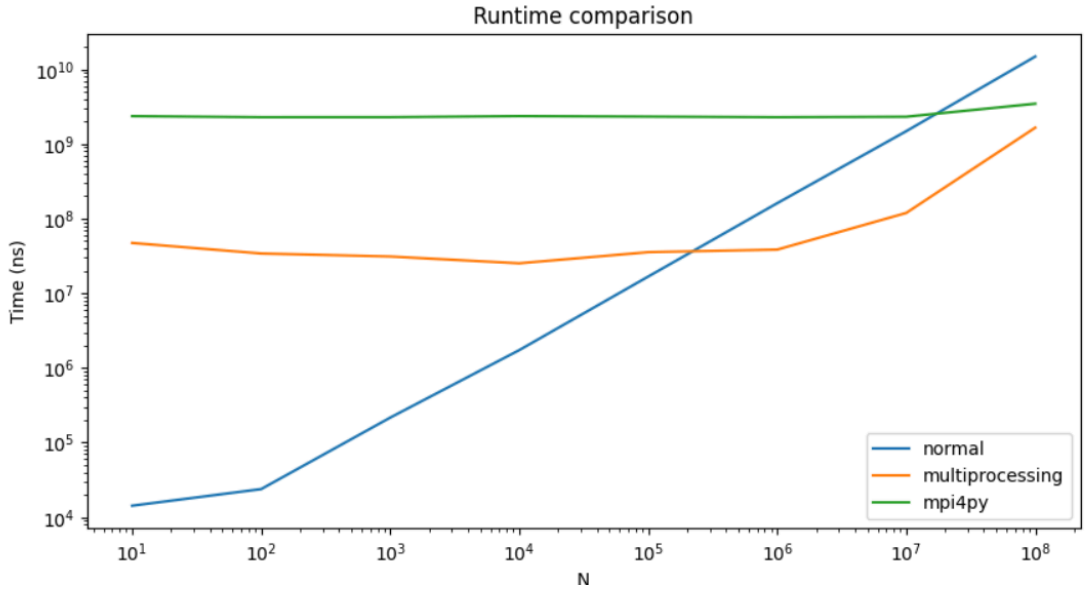
```
s = approximate_pi(100000) * 4

if MPI.COMM_WORLD.Get_rank() == 0:
    print("pi =", s)
```

When talking about the operations relating to the Riemann sum, the mpi implementation is the same as the multiprocessing one, its applying the parallel programming which is different here.Using the COMM_WORLD, which is the communicator, which can be seen as a grouping of all the processes and the interconnections that allow them to share information between them, the variables needed to set the size of the problem and which specific values will be handled can be obtained. Using the size of the communicator gives us how to do this division, and using the rank of each process just like the id in the multiprocessing implementation. All the local sums are added together using an Allreduce operation, which reduces the values and gives the result to all processes, in this case, using the sum operation. Finally, only one process prints the result out. To run this one, the mpiexec command is needed and is ran as follows:
mpiexec -n 8 python pi_approx_mpi.py

The important part of this command is the n flag, which defines how many processes will be used (8 were used to align with the CPU count previously used)

Profiling

The profiling was done in a Python notebook; for the normal implementation and the multiprocessing implementation, the time module was used, using the time_ns function, and for the mpi implementation, the time command from Unix was used, and the results were converted to nanoseconds. The tests were done starting with an N value of 10 and adding a 0 until the value of one hundred million. The following graph was created, with both the time axis and the N axis on a logarithmic scale so that the comparison is clearer.



While the normal implementation starts with much better performance, probably due to the overhead of using parallel programming for a size of problem that does not require it, it completely loses against both implementation, the multiprocessing module being the better one, which starts gaining performance after N = 10^5, and for mpi4py after N = 10^7. While the multiprocessing module has better performance than mpi4py in the entire graph, mpi4py clearly starts gaining some performance against it.

Conclusions

It is very clear that for problems of increasing size, normal implementations lack a lot of performance, clearly showcasing the power of parallel programming.

The multiprocessing module with its creation of concurrent processes creates a great boost of performance, and while its many functions come in handy, it is very surprising how its official documentation only explores that functional side, since using extensions of the Process class to instantiate the concurrent tasks gives a more custom and dynamic approach to defining how exactly each of them runs.

Another important conclusion is that these results do not imply that mpi4py is worse even in the smaller sized problems; MPI stands for Message Passing Interface after all, and the name already shows that the strength of this module is its communication system; using it in a laptop with shared memory will not showcase its strengths, but on a distributed memory system, this library is definitely more convenient and most likely, more performant and easier to implement than the multiprocessing module.

Code running evidence