Alex Funnell
DofE Bronze
Volunteering
25 February 2020

# Parallel Computing with MPI

## Introduction

My volunteering module of DofE Bronze has been with the AGS IT Department. This is the second section where one of my tasks was:

### Use four Raspberry Pi's in a cluster to calculate π to the highest number of digits in an hour.

I have also researched the concepts of cluster computing as part of this. Other information about the project is as follows:

- All research will be available for public use, including concepts and guides.
- Message Passing Interface (MPI) will be used as the basis.
- All nodes will be running Ubuntu Server.

## Research

### Concepts

Introduction to cluster computing

Cluster computing is a type of computing in which a group of computers are linked together so that they can act like a single computer.

In its most basic form, a cluster is a system comprising two or more computers or systems (called nodes) which work together to execute applications or perform other tasks, so that users who use them, have the impression that only a single system responds to them, thus creating an illusion of a single resource (virtual machine). This concept is called transparency of the system.

How cluster computing works

There are **two** types of nodes in a cluster. A *controller*, which distributes the tasks and controls the cluster and *workers*, which do what they say on the tin, carry out the task. Controllers are sometimes called master nodes or governing nodes. A computer in a cluster is known as a **node** whether it is a controller or worker.

Tasks are distributed evenly across the nodes so that they can be ran with multiple processors. Tasks also have to be written and designed in a certain way that uses the cluster.

Our nodes will run Ubuntu Server as the operating system and use Python as the high-level language that we will write the tasks in.

## Finding primes in parallel

Introduction

prime.py is a Python task that calculates prime numbers up to a certain endpoint over a single or multiple processors in parallel. I found some sample code on MagPi for doing this and added my own modifications. This was the first major milestone in the project. It enables me to then move on to researching solutions to the final objective of calculating pi in parallel.

Dependencies

1. mpi4py
2. time
3. sys

Prime.py Source Code

```
from mpi4py import MPI
import time
import sys
#from tqdm import tqdm

# Attach to the cluster and find out who I am and how big it is
comm = MPI.COMM_WORLD
my_rank = comm.Get_rank()
cluster_size = comm.Get_size()
```

```
# Number to start on, based on the node's rank
start_number = (my_rank * 2) + 1

# When to stop. Play around with this value!
end_number = int(sys.argv[1])

# Make a note of the start time
start = time.time()

# List of discovered primes for this node
primes = []

# Loop through the numbers using rank number to divide the work
for candidate_number in range(start_number,end_number, cluster_size * 2):

    # Log progress in steps
    #print(candidate_number)

    # Assume this number is prime
    found_prime = True

    # Go through all previous numbers and see if any divide without remainder
    for div_number in range(2, candidate_number):
        if candidate_number % div_number == 0:
            found_prime = False
            break

    # If we get here, nothing divided, so it's a prime number
    if found_prime:
        # Uncomment the next line to see the primes as they are found (slower)
        #print('Node ' + str(my_rank) + ' found ' + str(candidate_number))
        primes.append(candidate_number)

# Once complete, send results to the governing node
results = comm.gather(primes, root=0)

# If I am the governing node, show the results
if my_rank == 0:

    # How long did it take?
    end = round(time.time() - start, 2)

    print('Find all primes up to: ' + str(end_number))
    print('Nodes: ' + str(cluster_size))
    print('Time elasped: ' + str(end) + ' seconds')

    # Each process returned an array, so lets merge them
    merged_primes = [item for sublist in results for item in sublist]
    merged_primes.sort()
    print('Primes discovered: ' + str(len(merged_primes)))
    # Uncomment the next line to see all the prime numbers
    #print('Primes found:\n'+str(merged_primes))
```

## Calculating π

After many hours of researching and evaluating potential algorithms or series to calculate π, I finally came to the conclusion that there appears to be no benefit in using parallel processing to calculate π because of the mathematical methods needed to derive π. It is more efficient to calculate π using sequential programming methods and therefore I wouldn't use a Raspberry Pi cluster or any parallel computing methods to perform the calculation

The series that I researched were:

- Leibniz's Series $1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}.$
- Nilakantha's Series $\pi = 3 + \frac{4}{2 \times 3 \times 4} - \frac{4}{4 \times 5 \times 6} + \frac{4}{6 \times 7 \times 8} - \frac{4}{8 \times 9 \times 10} + \cdots$
- Spigot Algorithms
  - Algorithms used to compute irrational (non-repeating) numbers such as π or $e$ digit-by-digit from left to right with limited amounts of middle storage.

### Why can't you use parallel computing to calculate π?

**The short answer:**

That's not how it works. Computing the digits of π is like building a skyscraper. You cannot just assign different floors to different contractors to build at the same time and combine them at the end. You need to finish each floor before you can build the floor above it. The only way to parallelize is to have different contractors work together within each floor. The parallelism is horizontal, not vertical. The calculation of π is sequential and cannot be done in parallel.

**The longer answer:**

At the very top level in the calculation, it is not parallelizable at all. All the parallelism is at the lower levels. Therefore, communication between all workers is very frequent - more so than if it was on a single computer on its cache. This is enough to become a bottleneck of the task and slows the calculation down significantly.

Algorithms exist that directly compute random digits without the memory cost of computing all the digits before it. These are called "digit-extraction" algorithms.

However, these algorithms require an exponential time to compute a small number of digits after an offset of $N$ digits. Using this approach to compute all the digits from 1 to

$N$ will result in a quadratic algorithm. This makes it unsuitable for large numbers of digits.

**However...**

I have code to calculate π sequentially on a single computer. This code uses a Spigot Algorithm to work out π digit by digit, although as I explained before, this gets exponentially more difficult the more you go on.

For example, I was able to calculate around 11,000 digits in 10 seconds, but only around 38,000 in 120 seconds on my i7 MacBook Pro.

It's logical that it is actually faster to calculate π on a single computer rather than a cluster with this algorithm. This is because the calculation for a digit requires parameters from the previous calculation, about 10 variables. In practice, this is a very small amount of data, so when the computer executes it, it stores these values in the CPU cache rather than memory. The speed of reading/writing data from the CPU cache is extremely faster than doing the same through the relatively slow BUS route to the memory. If you wanted to parallize this task, it would be even slower as it would have to cross through the Ethernet, Switch and network adapters etc.

Calculate π Source Code

```
import multiprocessing
import time
import sys
from tqdm import tqdm

def calcPi():
    q, r, t, k, n, l = 1, 0, 1, 1, 3, 3
    while True:
        if 4 * q + r - t < n * t:
            yield n
            nr = 10 * (r - n * t)
            n  = ((10 * (3 * q + r)) // t) - 10 * n
            q  *= 10
            r  = nr
        else:
            nr = (2 * q + r) * l
            nn = (q * (7 * k) + 2 + (r * l)) // (t * l)
            q  *= k
            t  *= l
            l  += 2
            k += 1
            n  = nn
            r  = nr

i = 0
pi_digits = calcPi()
startTime = time.time()
duration = 120
```

```
endTime = startTime + duration
validTime = True


for digit in pi_digits:
    sys.stdout.write(str(digit))
    i += 1
    currentTime = time.time()
    if currentTime == endTime or currentTime > endTime and validTime:
        print('Digits found:',str(i))
        timeTaken = currentTime-startTime
        print('Took',timeTaken,'seconds.')
        validTime = False
        exit()
```

## Conclusion

After realising that the task wasn't possible, I finished having not completed the task as was exactly requested. The documentation shows how to set up raspberry pis in parallel. However, I learnt a lot with this project about cluster computing and the concepts around it and also about how to calculate pi.  The entire project is on Uncast's DevOps at this link:

https://dev.azure.com/Uncast/Quark/

The project includes this document, all source code and documentation with guides and tutorials and how to connect multiple Raspberry Pis in parallel clusters. This will be available for all students at AGS to refer to, and the public.

I really enjoyed working on this project, although there were definitely some trip-ups, and a lot more hours put into it than I expected, and I hope other students will find the project useful & interesting for their own needs and learning.

Commits: 80 (at time of writing)

Hours: 20 (estimated)

Cups of coffee: 18

**Written by Alex Funnell (@vulcno)**

**Student at Aylesbury Grammar School, Founder of Uncast.**