ID: 10575639
MODULE: PRCO304

PRCO304

Cluster Computing on Low Power Microprocessors

Words: 9242

Author: Erin Clifton

Contents

1. Introduction

1.1) Modern Parallel Computing

Processors with parallel capabilities have been in use since the 1950's, however, it is only recently that they have begun to impact commercial and domestic computing. The first commercial multithreaded processor was developed in 1968 but was never released to the public and the first domestic processor with multithreading was the Pentium 4 line from Intel, released in November 2000 and supported all the way to 2008. Now, it is incredibly difficult to find a newly produced processor without multiple cores. Most chip makers realised that it was impossible to keep single core processors competitive with multicore processors, with the end of Moore's Law seeming to be the final nail in the coffin (Crowell 2010). This uptake of multicore processors has even made its way to some IoT devices with all but the first generation of Raspberry Pi's featuring Quad Core Broadcom processors and newer BeagleBone boards featuring Dual ARM Cortex-A15 chips (Upton 2015).

CPU's (Central Processing Units) are not only the processor capable of running parallel code. GPU's (Graphics Processing Units) have been used since the late 1980's to run logical code on, with the first using the graphics chipset on an Amiga Desktop where it ran a genetic evolution algorithm referred to as "The Game of Life" (Hull 1987) approximately 300 times faster than a previous implementation on time relevant hardware. These processors can be referred to as GPGPU's, General Purpose Graphics Processing Units. The hardware for GPU cores usually mimicked the logical steps for rendering with its API up until 2006 where NVIDIA implemented the Unified Shader Model where all of the steps in the shader pipeline had the same capabilities, generalising their purpose in the GPU (Unknown Author 2018). Each of the cores had access to the same instruction set and to the same resources as each other. Ideally this would allow the card to load balance based off of software needs but it also opened the path to allowing general purpose computations to run on conventionally graphical focused hardware.

1.2) Conventional Parallel Architectures and Hardware

One of the best known architectures for parallel computing is the NVIDIA CUDA framework. First introduced in 2007, it provided a universal API for developers to perform general purpose computing on GPU hardware. Developers using CUDA adopted what is known as the CUDA processing flow (Zeller 2011), which are the basic programming steps needed to perform processing on the GPU, into their development style and is as follows: Copy data from RAM to GPU Memory, initiate the GPU compute kernel (the function being evaluated), execute the kernel in parallel, copy the result from GPU Memory to RAM. Having a common development style not only allows code to shared and understood far more easily but provides a basis for newer developers to grasp the new concepts provided by the CUDA framework.

CUDA programs can only be run on CUDA supported NVIDIA hardware, however, the processing flow can be used as guidelines for writing parallel programs in many other modern scenarios. Other frameworks for parallel processing include OpenCL, a highly popular generalised API for running computation on graphics hardware; AMD's Stream SDK, which includes an AMD hardware specialised version of the C programming language, a software interface for low-level hardware access and support for OpenCL 1.1; and mCUDA, a translation framework that compiles CUDA kernels to run on a CPU architecture, written and maintained by Andy Schuh of The Impact Research Group from the University of Illinois.

1.3) Parallel on Low Power Devices

Standalone low power devices are usually unsuited to parallel tasks due to the average small core count, lack of hyperthreading technology, and low core clock speed ceiling (Moon 2016). However, this low power, low speed processing is why this investigation is being done. The Hypothesis for this investigation is that low power devices can be used for parallel computation by simulating the cores in a multiprocessor. Thus, rather than using a low power device by itself to run a parallel program, a “cluster” of many low power devices can be linked together to simulate a processor with a large core count. This has been theorized to have been in practice as long as computers themselves as a means to overcome a workstations maximum capacity (Pfister 1998). Each individual device remains as powerful as it was before but could instead share its workload. Instead of a single device performing a million calculations, 10 moderately slower, but far lower power, devices could perform 100,000 each faster than the single device could the million computations. This has the potential to save vast amounts of time and money through electricity and setup costs. A high end NIVIDIA CUDA graphics card can use in excess of 290W whereas most low power microprocessors use an average of 5W. Whilst no amount of low power microprocessors can rival the computing power of high end NVIDIA Quadro GPU’s, they could possibly provide a far cheaper sufficient alternative.

The raspberry Pi foundation has made incredible strides towards this goal in recent years, creating and revising dedicated compute modules that can be slotted into custom PCB’s (Raspberry Pi Ltd 2016). Each compute module provides all of the resources and IO channels of its classic counterpart but is able to be slotted into a larger machine. Each of the Compute modules functions as a separate machine but can be linked viable available data pins.

However, this requires custom built PCB’s which are time consuming to design and are usually incredibly specific in use. This report aims to investigate the use of general purpose low power devices for a wide array of uses which is why these boards are not explored in more depth.

One of the limiting factors for linking low power devices is the maximum possible network traffic. Whilst a conventional NVIDIA Graphics card has in excess of 4000 CUDA compute cores that can be accessed for parallel computing, linking even a tenth as many low power devices would create incredible network traffic. Linking via a mesh network would draw vast amounts of the already limited processing power towards receiving and rerouting data not intended for the device.

Using a conventional Ethernet or Wi-Fi network also introduces latency that could be problematic for time sensitive operations such as real-time video or image processing. Even a 10-20ms delay due to network latency could cause lag, and without proper lag compensation techniques, could spell disaster for real-time operations.

This is not a nail in the coffin for low power devices as a parallel option however, as a CUDA compute core has only a fraction of the power and resources of a conventional CPU. Microprocessors can perform far higher power computations individually. A more measured approach is required rather than simply breaking a problem into its smallest possible components. The phrase “Serial in Parallel” would be apt to describe the coding style that would best suit a Low Power device cluster: executing concurrent serial code on Parallel hardware.

2. Hardware Choices

To properly investigate the use of low power devices for parallel tasks, a suitable microprocessor needs to be chosen.

2.1) MoSCoW Requirements

For this investigation, the requirements were fairly relaxed. The main “Must” requirement is that the board has networking available to be able to share tasks, data, and telemetry. The backbone of this project is the board’s ability to communicate effectively.

Must	Should	Could	Would
Have over 1Mbps Networking capabilities	Be smaller than mATX Form Factor	Have a built-in sensor array	Have a built-in battery Pack
Have less than 5W power draw per core	Be cheaper than £40		
Have at least 256MB of RAM	Be widely available		

2.2) Available Choices

There are near countless low power general purpose microprocessors. The options displayed here are just a few of the most popular and widely available boards with networking capabilities.

2.2.1) BeagleBone Black - http://beagleboard.org/black

These low power microprocessors were the first board investigated. They have a 100 mbps Ethernet port, can run off of a standard USB power only connection and have 1GB of in built RAM. Another incredibly useful feature is the presence of on board EMMC storage which facilitates the easy setup of new boards. By flashing an already made image to the onboard storage, the setup time can be effectively quartered.

This board is unfortunately just over the £40 threshold, but it is available from a large number of distributors and is the size of a credit card. It has neither a sensor array nor a battery pack, but it has 46 header pins which can be used to set up a whole host of possible sensors.

There is an unfortunate bug on the last stable release of developer intended OS image. The ethernet host will very often need to be reset on boot before connections can be made. This removes the board from consideration as the server boards must require little maintenance. Individually resetting each board is simply an unacceptable cost.

2.2.2) Raspberry Pi 0 W - https://www.raspberrypi.org/products/raspberry-pi-zero-w/

These boards are incredibly cheap, costing just £6 per board; have an incredibly low power draw of 5V 200mA at maximum draw; have Wi-Fi capabilities; are about as small as a USB memory stick; have 512MB of RAM, and are sold by several companies.

These boards would be the ideal board for this project; however, none were available from any supplier for at least 2 weeks at the start of this investigation. Rather than stall the whole investigation for the possibility of a restock, a different board was chosen.

2.2.3) Arduino - https://store.arduino.cc/arduino-yun

The Arduino Yún Rev 2 would have been the board of choice for an Arduino. It has both Ethernet and in-built b/g/n Wi-Fi. This version has a Linux Microprocessor alongside its usual microcontroller. It can be run off of a 5V 1A USB supply and is slightly smaller than a credit card. It also comes with 32 Header pins for hardware extension.

However, there is only 64MB of RAM, and a worryingly low clock speed of 400MHz, far lower than any other board investigated, even the Raspberry Pi 0, which has a clock speed of 1GHz. This led onto the final choice.

2.2.4) Raspberry Pi 3 - https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/

The Raspberry Pi 3 Model B+ has a quad core 1.2GHz ARM processor, 1GB of RAM, in-built Wi-Fi, a 100 mbps Ethernet connection, a dedicated camera port, and 46 GPIO pins for extensibility.

It has a rather high power draw of 5V 2.4A, but this is powering 4 cores and Wi-Fi. If all 4 cores are used for computing, then the average current per useful core is only 600mA. The board only costs £37.

2.3) Final Choice Suitability

The final choice for this investigation was the Raspberry Pi 3 B+. The model 4 offered no real value over the 3 save for a slightly faster clock speed. The 3 B+ can effectively be viewed as a cluster of 4 small microprocessors for the purposes of this project. Each core can be used individually for computation and as such can be treated as a separate device for running computations. This makes the 3 B+ 4 times as effective as it would first seem.

The use of ethernet allows 16 cores to connected using only 4 cables. This massively reduces network traffic which can become a problem when dealing with high volumes of data. A Powerpack can also be used to supply all the boards from USB to Micro USB power cables. This increases the portability of the project. Instead of a row of mains chargers, a single AC power cable is used to power the entire project.

3. Software Design

3.1) Language Choices

3.1.1) C++

The C++ Language offers incredible control over memory and functions. All but the Arduino board support this language. C++ development is far more involved than other languages, due to the incredible control it provides over memory management, and so would take far longer to implement. For fully optimized code, C++ would be the choice, however, this investigation aims to prove the possible use of Low Power devices for Parallel Tasks, not to find their absolute limit. Therefore, a simpler language was chosen.

3.1.2) MATLAB

MATLAB can be used to great effect when controlling data and performing high complexity computation to it. The raspberry Pi, and to a lesser extent the BeagleBone Black, support the use of MATLAB. Development of MATLAB code is fairly swift as it offers a lot of in-built functionality to support development and testing. However, the IDE and compiler are very resource intensive, featuring a fully fleshed IDE and compiler. It can be a struggle to run even for the relatively powerful Raspberry Pi 3 B+. For this reason, MATLAB was not used.

3.1.3) Node Js

Node is a powerful runtime of JavaScript that can be used to perform networking tasks easily. Using Node an API could be set up to facilitate transfer of data between the main board and the compute boards. However, this would require the use of a board as a sole API host or host the API on the main board and risk severely reducing its computing power. This setup would also need another language for the main process and compute processes as writing JavaScript for machine learning would be arduous at best. For these reasons, Node Js was not chosen.

3.1.4) Python

Python ships natively on most Linux OS images. It offers many options for data control and networking, with many third party libraries to choose from to extend its capabilities. One third party library in particular, Dispy, provides framework for cluster computing. Whilst the runtime can be slow for simple tasks due to Pythons lack of array class, Python offers many options for optimization. For these reasons, Python was chosen as the main language for the project.

3.2) Networking Layer

For cluster computing, the individual processors must be able to communicate with one another. There are several options for frameworks to send and receive data.

3.2.1) MQTT

MQTT is a lightweight publish/subscribe IoT connection protocol. Its main use is "for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium". For small devices with a split workload, this can be incredibly useful, especially as there is an MQTT library implemented in python. However, it lacks the ability to send messages directly to recipients, instead relying on a broadcast publish/subscribe model. When published, every subscriber receives the same message. This makes it difficult to assign workloads effectively and would require intensive load balancing and a method of distinguishing which message is meant for which recipient.

3.2.2) Zigbee

Zigbee is an 802 Wi-Fi specification designed for IoT Devices. It acts as a communication standard rather than a networking layer, meaning that out of the possible hardware, it its only useful for Wi-Fi on the Arduino. For all other devices, ethernet or standard 802.11 b/g/n Wi-Fi is sufficient. It would also only be part of the networking solution, as a data transfer protocol would still be required.

3.2.4) Dispy

Dispy is a third party python library that implements several classes and methods to aid in cluster computing. It operates off of a "One client to Many Server" approach.

The client script designates a cluster object with a list of IP addresses of the server nodes and a python function for the nodes to run. The client script then "submits" jobs to be done to the cluster. These jobs are then allocated to the nodes depending on their current workload. The client then calls the job() function for each submitted job to the cluster to collect the results. The functions supplied run in isolation. No imported modules are called and there is no access to global variables.

This approach allows any number of nodes to be added to the network, and the load balancing ensures that both: jobs are not sent to overworked nodes and nodes are not left without work to do whilst jobs are left to be allocated.

The server script is able to run on multiple threads which allows it to take advantage of multiple cores. This is excellent for the hardware choice of the Raspberry Pi 3 B+ as the board has a quad core processor.

3.3) Possible Parallel Tasks

Once the hardware and base software is set up, the cluster needs a problem to solve to showcase any potential it may have. There are countless possibilities with varying degrees of possible parallelisation. Some tasks are more suited than others to this architecture and the "Serial in Parallel" coding style that was used to develop code for the project.

3.3.1) Machine Learning

There are an incredible amount of machine learning algorithms available. The main 2 algorithms that could easily and realistically be implemented on this hardware are the K Means classification and the K Nearest Neighbours classification. To gain a useful base line for this projects suitability to machine learning the K Means algorithm was used.

The K Means classifier works by sorting data by its mean distance to the centre of each classification. In practical terms, K different classifications are chosen. The first K data points are assigned as the centre of each of the K clusters. Each datapoints distance to each of the cluster centres (centroids) is then calculated by squared Euclidian distance. The data point is then assigned to the closest centroid. Once all data points have been classified, the centres of the clusters are recalculated as the mean of features of each data point in the cluster.

The process of classification then recalculation is repeated a specified number of times, or until there is insignificant variance in the changes of the centroids.

Once the cluster centroids have been found, the first step of classifying each data point, the centroid mapping function, can be used to classify new data. The centroids can be recalculated after each step, but this is not necessary.

The transition from serial to parallel with this algorithm is relatively simple, as the classification function can be run independently of the data before or after it, which allows it to run in an isolated environment. This is perfect for the dispy framework, as the nodes create an isolated python environment for the function to run.

3.3.2) Advanced Mathematics

There are countless incredibly complex tasks that can be performed on parallel hardware. For the purposes of this investigation, the Taylor Series was chosen as it provides a promising basis from which to parallelise its functions. The Taylor Series of a function can be described as an infinite series, the sum of which is equal to value of the function it describes. Each value in the series is calculated in terms of the functions derivatives at that point. In algebraic terms, $f(x) = \int_0^x f'(t)\,dt$ which can be re-expressed in terms of x as a series unique to the function.

The function chosen for this was the Arctangent function which applies in the inverse of the tangent function. $arctan(tan(x)) = x$. However, as the tangent function repeats itself, the range of the arctan function is limited.

This infinite series provides a solid means of parallelisation as the calculation of each term of the series is independent of the term before it, allowing multiple consecutive terms to be calculated at once.

3.3.3) Parallel Search

For high volumes of data, searching can take an incredibly long time, especially if the data is complex or unsorted. Parallelising a binary search algorithm could provide incredible speed up by split the dataset into smaller chunks that take far less time to search than performing the function concurrently.

3.3.4) Generalised Elementwise Operations

One of the drawbacks of the dispy framework is that every time a different function needs to be computer in parallel the cluster must be reassigned which could take precious time. Implementing a general function that could take a different function as a parameter and perform that function on the other arguments could save valuable from reassignments. For this reason, this task was chosen to demonstrate on the hardware.

4. Implementation

4.1) Hardware

The hardware architecture of the project is a single client, multiple server setup consisting of 5 Raspberry Pi 3 B+ Boards in a server stack case, with a small 5 Port desktop switch to create an ethernet network. Power is supplied to all devices via a 6 Port USB Power Pack providing 5V 2.4A to each USB port, supplied by a single 13A AC mains cable. As every device can be powered by a single cable, this allows the project to be portable with easy setup and shutdown.

During development, a name was needed so as to be able to discuss the project and compare it to similar computers. Due to the layered nature of the boards and their food like name, the

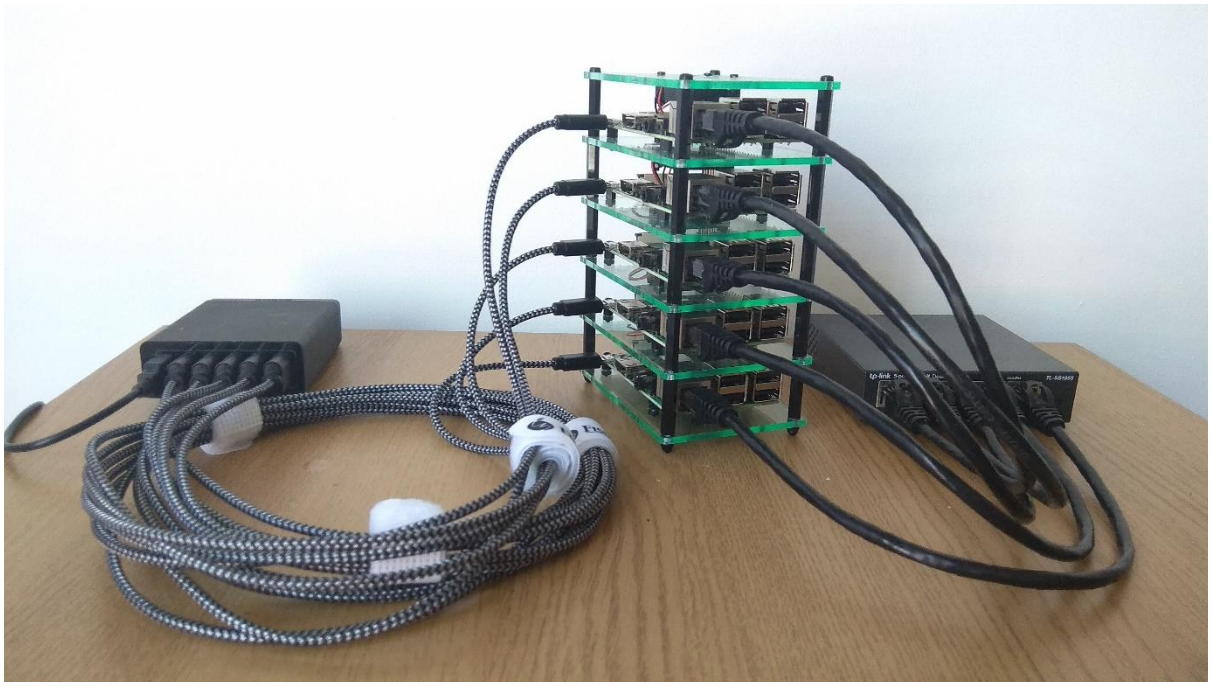


Figure 1 - Project Lasagna Hardware

stack was informally dubbed “Project Lasagna”. For the purposes of this investigation it will be referred to as either “The Project” or “Project Lasagna” interchangeably.

The cluster itself is composed of 5 Raspberry Pi 3 B+ boards in the light stack case available from the Raspberry Pi Foundation. Each case comes with a 2 pin fan, a header and bottom board and header board to protect the Pi. This style of case is easy to extend as more cases can be added to the top or bottom of the stack. Each of the Pi’s is plugged into both the Power Pack and the Network switch, the order of the cables is not important here other than careful cable management. The last open USB slot open on the Power Pack is used to power the network switch.

To set up the server boards, the server microSD cards were loaded with the latest Raspbian Buster image from the Raspberry Pi foundation. This image comes preinstalled with python and several large python libraries. The Dispy framework was installed using the Pip3 python module manager. This comes with a server file that can be run from command line with the devices IP address as an argument to start running the Dispy server node. A short bash script can be created then added to crontab to have the node start when the device boots. As the node script requires the devices IP address as a starting argument it is advised to enable the option in the raspi-config menu to wait for network on boot. At this point, an SD card duplicator could be used to save time setting up multiple devices. Due to cost constraints, this method was not used in this investigation.

For the client board, the setup was fairly similar to the servers. A microSD card was loaded with the latest Raspbian Buster image and the Dispy framework was installed. The setup then diverges from the server boards. An SSH key was generated then shared with the server boards to allow remote access for shutdown and reboot commands. Whilst Raspberry Pi’s themselves are not damaged by simply pulling out the power cord to cut the power, the SD cards can become corrupted if data was being written at the time. From this point, python code can be written by importing the Dispy library to a python file.

To ensure that the server boards can communicate over an unmanaged ethernet network each of the 4 boards was given a unique static IP address in the range 10.0.0.3-6, with 10.0.0.2 reserved for the client board. This can be done by enabling the dchpcd service if not already enabled, then editing its config file to allocate a static IP. This static IP is important for Dispy too, as the IP address of the server nodes are used as an argument to create the cluster object. Whilst an IP address with wild card values (i.e. 10.0.0.*) can be given if IP address are known only by their local subnet mask or allocated dynamically via DCHP, it marginally speeds up the process of finding the nodes to give specific IP Addresses.

4.2) Serial Implementation

Each algorithm investigated was first developed as a serial function that could be called using an arbitrary amount of data. This method of development allowed the function to be used for parallel purposes with little direct editing, the “Serial in Parallel” approach. The main body of processing for the algorithm is done within the functions defined at the top of each file. The rest of the code is used as an example of how the function is used and will not be shown in this report.

For the purposes of commenting a double hash (##) is for an explanatory inline comment, a single hash (#) is for a commented out code option

4.2.1) K Means

```
def cmap(mapCen, mapData):
    mapResults = []
    numFeatures = len(mapData[0])
    ##for each data piece
    for dataToMap in mapData:
        eucDists = []
        ##iterate through each centroid
        for cen in mapCen:
            ##find the Squared Euclidean Distance to the data piece
            SED = 0
            for feature in range(numFeatures):
                dif = cen[feature] - dataToMap[feature]
                SED = SED + (dif * dif)
            eucDists.append(SED)
        ##record the centroid with the shortest distance
        mapResults.append(eucDists.index(min(eucDists)))
    return(mapResults)

def sortDataToCluster(dataToSort, dataIndicies, k):
    clusteredData = []
    ##set up empty results array
    for cluster in range(k):
        clusteredData.append([])
    ##dataIndicies[d] returns the centroid ID of the data stored in
dataToSort[d]
    for (d in range(len(dataToSort))):
        clusteredData[dataIndicies[d]].append(dataToSort[d])
    return(clusteredData)
```

These 2 functions make up the bulk of the K Means algorithm. The cmap function is especially important as it “maps” each datapoint to the closest centroid. This function is also used to implement a trained centroid list. When a datapoint needs to be classified, it is run through the cmap function with the list of trained centroids to find the cluster it fits best with.

The sortDataToCluster function was initially implemented as inline code but was moved as the section returns a single object and also to increase ease of reading the code.

4.2.2) Taylor Series Expansion (Arctan)

```
def taylorarctan(indicies,x):
    sum = 0
    for n in range(indicies[0],indicies[1]):
        ##precalculate middle step
        middle = ((2*n)+1)
        sum = sum + ((-1)**n)*((x**(middle)) / (middle))
    return sum
```

This function performs the Taylor Series expansion for the arctan function as laid out by Spiegel, M., (1968). It takes 2 arguments, a pair of indicies in a list and the value x for f(x) = arctan(x). The pair of indicies refer to a section of the Taylor Series expansion to be computed. A middle value of (2n+1) is precalculated to facilitate ease of reading the code. The sum of the partial series is calculated then added returned to be added to the whole answer.

4.2.3) Binary Search

```
def binSearch(tar, searchList):
    results = []
    for i in range(len(searchList)):
        if (searchList[i] == tar):
            results.append(i)
    return results
```

The binary search algorithm is incredibly simple. It checks each value in a list in turn against the supplied value to search for. When it finds a value that matches, it adds that values index to a list of results then returns that list once all values have been iterated through.

Parallelizing this is as simple as splitting the data to search through into multiple pieces and recombining the results into a single list. It is worth noting that the function returns the local index of the result and not its position in the larger dataset. To make use of this the index can be translated to point to the whole set or the index can be used to directly access the result from the partial set if it still exists.

4.2.4) Elementwise Operations

This section of code is an example of the limits of the dispy framework. The idea presented here is to create a general function that takes a lambda function and either 1 or two datasets then applies the lambda function to the provided data. This would allow a developer to perform multiple different tasks with the same cluster. However, Python is a pass-by-object-reference language. This implementation would only work in a serial environment. When run in parallel, when a job is sent to a node, the function argument provided is an object reference that the node cannot understand. The function was only initialised on the client, so the node does not know what the reference references. Because of this the cluster will hang and eventually return an error after the timeout period elapses.

```
def performElementwiseFunction(func,dataset1,dataset2=None):
    results = []
    if not(dataset2):
        for data in dataset1:
            results.append(func(data))
    else:
        import itertools
        zipData = zip(dataset1,dataset2)
        for data in zipData:
            results.append(func(data[0],data[1]))
    return results
```

4.3) Parallel Implementation

Because the main bulk of the algorithms in the serial implementations were defined into functions, the leap from serial to parallel only involves setting up the parallel environment and as such, the functions defined in the serial implementation are identical and therefore left out of the code displayed.

The basis of the Dispy cluster is the Cluster object. The cluster constructor takes the computation to be performed and the IP addresses of the nodes as arguments and returns a cluster object to which tasks can be added via the provided submit() method that takes the arguments of the desired computation as arguments.

Once a task has been assigned, the results are access via the job() function which implements a blocking wait until the oldest submitted task is completed, and then returns the result of the computation function as its return value. If a blocking wait in between each return of results is undesirable, the wait() method for the cluster object can be used to implement a blocking wait until all scheduled jobs have finished.

Any number of jobs can be sent to a cluster and can be submitted and collected in any order. If a node is to be used in a different cluster, e.g. a new computation needs to be executed, the existing cluster object must be closed.

4.3.1) K Means

Initially, both the feature averaging and centroid mapping tasks were implemented as parallel code. This previous implementation required reassigning the whole cluster each time the tasks at hands were switched. Each feature for every data point for each centroid was collected and sent off to be averaged. However, this task proved to be slower when parallelized, provided a speedup of 0.8-0.95 on average, 5-20% slower than it could be performed in serial. Only exceptionally large values for numbers of data points, features or a very large K would yield a positive speedup. Because of this the feature averaging is done client side in a serial manner.

The sortDataToCluster() method could possibly be parallelized via a form of merge sort, however, this would not provide a significant speedup as each merging each of the sorted lists would fairly intensive, and the current implementation of O(n) complexity is sufficient to sort fairly large data structures, i.e. 10,000 or more data points.

```
if __name__ == '__main__':
    import dispy, socket
    import numpy as np
    import matplotlib.pyplot as plt

    ##fetch the IP address of the client
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect(("10.0.0.2", 80)) ##doesn't matter if 8.8.8.8 can't
be reached
    ##read in data from text file, split into lines and turned
into floats
    f = open("data.txt","r")
    data = [[float(num) for num in line.split()] for line in
f.readlines()]
    f.close()

    numdata = len(data) ##length of dataset
    features = len(data[0]) ##length of a single data piece

    ##set k
    k = 15

    ##choose random centroids from available data
    centroidID = np.random.permutation(numdata)
    centroids = np.empty((k,features))
    for i in range(k):
        centroids[i] = data[centroidID[i]]

    ##repeat cycle x number of times to train centroids
    numtrials = 10
    numNodes = 16 ##usually total number of core on network,
maybe 2 times the number if feeling generous)
    numtrialtaken = 0

    for trialNum in range(numtrials):
        ## reset cluster for fresh use
        cluster =
dispy.JobCluster(cmap,ip_addr=s.getsockname()[0],
nodes=['10.0.0.3','10.0.0.4','10.0.0.5','10.0.0.6'])

        ##split each data piece into its own job
        numJobs = numNodes
        results = []
        jobData = np.array_split(data,numJobs)
        jobs = []
        ##new method writing direct into job list, however, it
doesnt allow for adding a jobID to the job
        #jobs = [cluster.submit(centroids,jobData[i]) for i in
range(numJobs)]

        for i in range(numJobs):
            ##schedule execution of 'cmap' on a node (running
'dispynode')
            ##with parameters (centroids of classes, batch of
data to be classified)
            job = cluster.submit(centroids,jobData[i])
            job.id = i
            jobs.append(job)


        for job in jobs:
            n = job() ##get job results
            ##.append would create a sublist. Dereferencing
arrays when combine prevents this
            if n is not None:
                results = [*results, *n]
            print('Executed cmap job %s in %s seconds' % (job.id,
job.end_time - job.start_time))
        cluster.print_status()
        ##MUST be called before new cluster defined
        cluster.close()

        ##sort data into clusters. This can be done as simple
assignment into a list, so parallelizing this would be worthless
        clusteredData = sortDataToCluster(data,results,k)
        ##clusteredData[class][data][feature]

        ##setup empty results
        results = []

        ##do feature averaging client side. Doing it server side
results in speedup < 1 due to network latency
        ##for each cluster
        for i in range(k):
            numDiC = len(clusteredData[i])
            ##for each feature in chosen cluster
            for j in range(features):
                featureSum = 0
                featureAvg = 0
                ##grab each value of feature from data in cluster
                for m in range(numDiC):
                    featureSum = featureSum +
clusteredData[i][m][j]
                ##if no data in cluster, use centroid instead
                if (featureSum == 0):
                    featureAvg = centroids[i][j]
                else:
                    featureAvg = featureSum/numDiC
                ##store result
                results.append(featureAvg)

        ##rebuild centroids from results
        newCentroids = centroids
        for i in range(k):
            for j in range(features):
                newCentroids[i][j] = results[(features*i) + j]
        ##check to see if centroids have changed
        compare = centroids == newCentroids
        if compare.all():
            ##if so stop classifying
            numtrialtaken = trialNum
            trialNum = numtrials - 1 ##exit the loop
        else:
            ##if not keep classifying
            centroids = newCentroids
```

4.3.2) Taylor Series Expansions (Arctan)

```
if __name__ == '__main__':
    import dispy, socket
    import numpy as np

    ##fetch the IP address of the client
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect(("10.0.0.2", 80)) ##doesn't matter if 8.8.8.8 can't
be reached

    ##Setup Cluster
    cluster =
dispy.JobCluster(taylorarctan,ip_addr=s.getsockname()[0],
nodes=['10.0.0.3','10.0.0.4','10.0.0.5','10.0.0.6'])

    ##Initialise Variables
    precision = 131072
    indicies = list(range(precision))
    numJobs = 16
    nums = (-0.9999999999, 0.9999999999, 0.0000000001,
0.7853194857, -0.82, 0.4792378463)
    joblist = np.array_split(indicies,numJobs)

    for x in nums:
        results = 0
        jobs = []
        for job in joblist:
            job = cluster.submit(job,x)
            jobs.append(job)
        for job in jobs:
            results = results + job()
    print(results)
    cluster.close()
```

4.3.3) Binary Search

Parallelizing the binary search algorithm is a relatively simple task. The task is trivial in its complexity, and the results are easy to recombine. The function can also be easily used for a contains search by interpreting the length of the results list where 0 is False and >= 1 is True.

```
if __name__ == '__main__':
    import dispy, socket, time
    import numpy as np

    # fetch the IP address of the client
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect(("10.0.0.2", 80))
    # doesn't matter if IP cannot be reached, only that the
socket exists

    cluster =
dispy.JobCluster(binSearch,ip_addr=s.getsockname()[0],
nodes=['10.0.0.3','10.0.0.4','10.0.0.5','10.0.0.6'])
    data = np.random.randint(0,4000,500000)
    numJobs = 16

    jobData = np.array_split(data,numJobs)
    jobs = []
    results = []

    searchTarget = 50

    for D in jobData:
        job = cluster.submit(searchTarget, D)
        jobs.append(job)

    for job in jobs:
        result = job()
        results = [*results, *result]

    print(results)
    cluster.close()
```

4.3.4) Elementwise Operations

For the parallel Generic Elementwise operations implementations, a different approach had to be taken. Dispy cannot handle the passing of normal or lambda functions as an argument to be used in computation, as python normally passes these by reference. For this reason, any generic function would instead have to emulate generality. The function provided for computation should contain any and all use cases for processing that must be done. The main benefit of such an approach when using dispy is that the same cluster can be used to perform several different computations. For a large cluster network this can save time that would be spend reassigning nodes with different computation functions. However, It would also greatly increase the amount of code management for large projects and multithreaded code, as every single use case for calling the generic function in the cluster needs to be considered and planned for to avoid unwanted results.

5. Comparison

In order for the concepts investigated by this report to have any significant use, the parallel implementation of at least 1 algorithm must show a significant speedup is a significant number of use cases. In many, if not all, cases a small speedup would not warrant the cost of maintaining a parallel cluster or the code that it uses.

To better understand how much time dispy takes to perform computations, a test function was developed as below. Each “job” sleeps for 1 second then returns its IP address. The calculated difference between what dispy reports and 1 second would be the time spent sending and receiving the job, i.e. the network latency. The difference between the dispy reported time and the python reported time could be due to the client side taking longer to process the results calling function than the node takes to perform the function. This is not necessarily evidence that task at hand is not suited for parallelization however, as there are still multiple nodes performing computations simultaneously.

The code below sends out 16 tasks to be completed. One for each core of the cluster, which is the ideal balance of jobs, any more and jobs would wait, any less and there would be free nodes. The task is simply to wait for one send then return its hostname. Each time a job is sent off the time is added to an array and once each job is returned, the time is taken again and added to a separate array for comparison. The overall time taken to compute the parallel function can be found by taking the time between sending the first job out and receiving the last job’s results.

```
def timeFunc():
    import time,socket
    time.sleep(1)
    return socket.gethostname()


if __name__ == '__main__':
    import dispy, socket, time
    import numpy as np
    ##fetch the IP address of the client
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect(("10.0.0.1", 80))
    cluster = dispy.JobCluster(timeFunc,ip_addr=s.getsockname()[0],
nodes='10.0.0.*')

    numJobs = 32
    starts = []
    ends = []

    results = []
    jobs = []

    for i in range(numJobs):
        job = cluster.submit()
        starts.append(time.time_ns())
        job.id = i # optionally associate an ID to job (if needed later)
        jobs.append(job)

    for i in range(len(jobs)):
        host = job()
        ends.append(time.time_ns())
        print('Dispy calculated %s executed job %s in %s seconds' %
(host, jobs[i].id, jobs[i].end_time - jobs[i].start_time))
        ##other fields of 'job' that may be useful:
        ##print(job.stdout, job.stderr, job.exception, job.ip_addr,
job.start_time, job.end_time)
        print('Real time taken calculated as: %s seconds' % ((ends[i] -
starts[i]) / 1000000000))
    cluster.print_status()
    cluster.close()

    print('Overall time taken: %s seconds' % ((ends[len(ends-1)] -
starts[0]) / 1000000000))
```

```
Dispy calculated raspberrypi-1 executed job 0 in 1.0162749290466309 seconds
Real time taken calculated as: 1.135161406 seconds

           Node |  CPUs |    Jobs |  Sec/Job | Node Time Sec |    Sent |    Rcvd
--------------------------------------------------------------------------------
  raspberrypi-4 |     4 |       4 |      1.0 |           4.1 |   665 B |   969 B
  raspberrypi-3 |     4 |       4 |      1.0 |           4.1 |   667 B |   971 B
  raspberrypi-2 |     4 |       4 |      1.0 |           4.1 |   663 B |   967 B
  raspberrypi-1 |     4 |       4 |      1.0 |           4.1 |   668 B |   972 B

Total job time: 16.430 sec, wall time: 1.213 sec, speedup: 13.545

Overall time taken: 1.199139844 seconds
```

As the results show, the dispy calculated time is on average, 30 ms over 1 second. This is most likely due to the importing of the time and socket modules and the last line of returning the hostname and is an acceptable margin for error. The difference between the dispy reported time and the python time module reported time is between 90-100 ms. This is far larger and cannot be put down to network issues alone as all devices are linked on an unmanaged 5 Port ethernet switch with an average ping of 12ms, and as can be seen in the results, less than a single kilobyte of data was transferred.

The reason for this divide is that between each task being sent out and received, every other task is also being sent out or received. Each task is sent out and received in order. Once the first job has been sent out, all other jobs must be sent out too before the first job is received. For the last job, all other jobs must be received before it too can be received. This small amount of processing and networking creates this apparent 100ms of latency.

Because of this, real time operations should only be sent out from a machine fast enough to handle the results. However, any machine powerful enough for that almost certainly has the hardware needed to process the operation itself.

The initial instance of testing and comparisons were done on a dataset of 100000 data points, each consisting of 2 floating point features, grouped around 15 centres. An exceptional dataset was generated with 7 centres and 2 features for display purposes and was used for validity testing. The code that generated this dataset is as follows.

```
import sklearn.datasets as dataset
numdata = 100000
features = 2
centers = 15
data = dataset.make_blobs(n_samples=numdata, n_features=features,
centers=centers)
f=open("data.txt","w")
for a in data[0]:
    tempstr = ""
    for x in a:
        tempstr = tempstr + str(x) + " "
    tempstr = tempstr + "\r\n"
    f.write(tempstr)
f.close()
```

Each feature is separated by a space and each data point is on its own line. Each line is written individually to prevent the excessive memory reassignment that comes from building large strings.

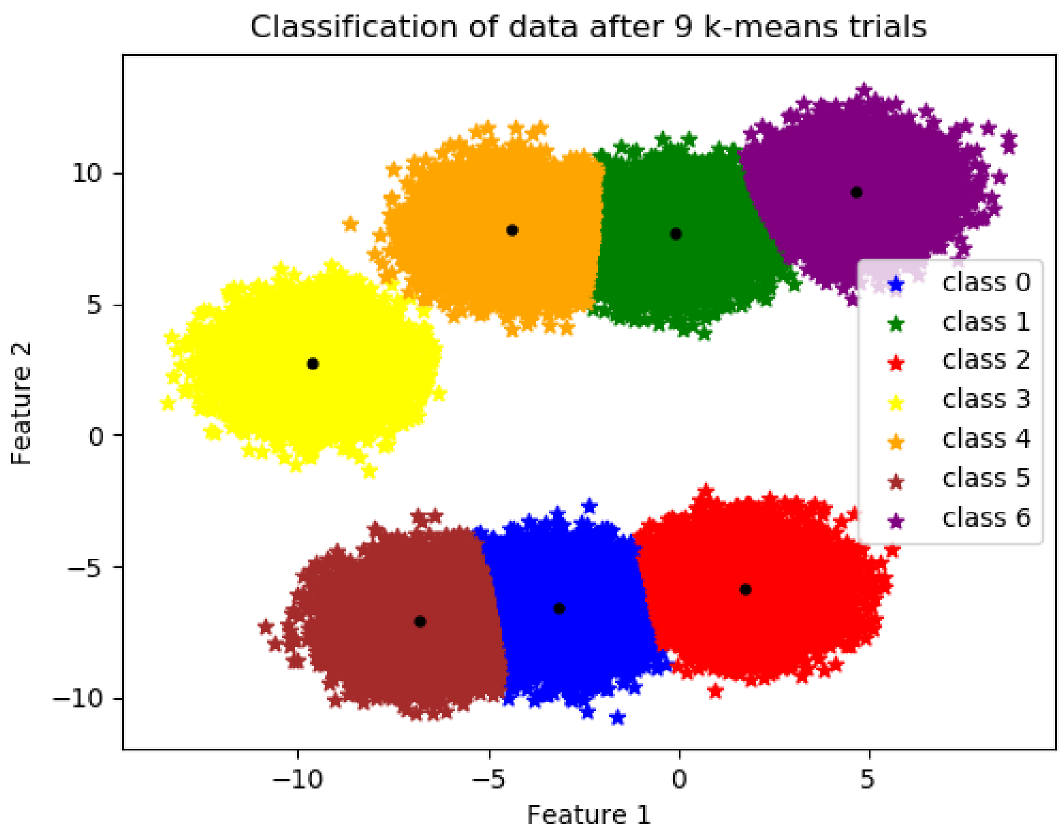
The make_blobs() function from sklearn was used to provide an obvious boundary between clusters for training K means and has no effect on the other algorithms.

A way of estimating the maximum possible speedup from parallelizing code is the application of Amdahl's Law: $S = \left(\frac{1}{1-p}\right)$. It works by comparing the runtime that can be parallelized to the runtime that is done in serial. P in the formula is the proportion of the runtime that is possible to parallelize as a decimal between 0 and 1.

5.1) K Means

The main difference between the serial and parallel implementations of K Means is the cluster implementation of the cmap (centroid mapping) function. Because of this, the bulk of the comparison will be focused on this function and how fast it can be run.

The K Means parallel algorithm was initially tested for validity on a dataset of 100000 data points with 2 features, with a K value of 7. The algorithm was given a maximum trial limit of 10, but was able to complete it 9, with the 9th providing no change in assignments of cluster centroid locations.



As can be seen above, the algorithm is clearly capable of classifying floating point data as it should.

All other tests were focused on the speed at which the training could be performed. These are the values used to implement a randomly created dataset.

All timings are collected in nanoseconds and converted to seconds.

Set Number	Number of Data Points	Number of Features	Number of Clusters (K)
1	100,000	7	15
2	200	2	2
3	1,000,000	16	25
4	100,000	4	4
5	200,000	4	4
6	100,000	8	4
7	100,000	4	8
8	200,000	8	8

The first dataset was used for initial comparisons at a basic level. A simple test of how fast the parallel implementation can run compared to the serial.

SET 1	Serial	Parallel
Test 1	80.3624	6.009
Test 2	82.4953	5.891
Test 3	82.8361	5.885
Average	81.8979	5.928

The second dataset is a test of both algorithms efficacy with a tiny dataset.

SET 2	Serial	Parallel
Test 1	0.0121 s	1.270
Test 2	0.0118	1.355
Test 3	0.0128	1.344
Average	0.0123	1.323

Here the dataset is so small that trying to parallelize it actively harms the performance with an average “speedup” of 0.09297 or approximately performing 9.3% as fast.

The third dataset is a test for the efficacy of each implementation with an extremely large dataset. A value of 4 million would have been used however, this proved to take far too large for the serial implementation to handle. After an hour of runtime, the implementation had yet to complete a single iteration of the K Means algorithm. Even with only 1 million data points instead of 4 the first test came out to 43 minutes of runtime. Due to this, only 1 test was performed.

SET 3	Serial	Parallel
Test 1	2604.1798	139.502
Test 2		140.422
Test 3		140.491
Average	2604.1798	140.138

The speedup for this large dataset is 18.583 times as fast. This is incredibly significant considering there are only 16 cores performing the operation. Individually, each core in the parallel cluster performed faster than the core running the serial implementation. One reasonable explanation for this is that the cluster was able to manage its memory accesses far more efficiently than the serial implementation. Rather than dealing with the entire 280 MB dataset, it was split into 16 different pieces. This makes the handling of the data in memory far faster to deal with, allowing the total computation to have greater per core efficiency. One work around for this would be to read the data straight from the datafile.

Each individual job took on average 40 seconds. With a job size of 32, each node had 2 lots of 40 second jobs to run totalling a runtime of 80 seconds which is far less than the reported time. However, with the 100mbps ethernet available to the pi and the dataset being 280MB, it takes a minimum of 22.4 seconds to transfer to the nodes when sending the job out and noticeably less time when receiving the indicies as results. Given how the single ethernet connection to the board is managing 4 different data streams which are all competing to transfer their data a time far greater than 22.4 seconds would not be unexpected as a delay.

The last 5 datasets are means to compare the effect on each implementation of doubling each of the important factors for K Means, with the first dataset providing a benchmark and the last dataset doubling each value (N, F and K) at the same time.

SET 4	Serial	Parallel
Test 1	15.6330	2.518
Test 2	15.6342	2.508
Test 3	15.5184	2.650
Average	15.5952	2.559

Speedup for this dataset is 6.094 times as fast.

This set is a benchmark for a sub investigation into the effects of altering aspects of the dataset into the efficiency of

SET 5	Serial	Parallel
Test 1	30.6650	3.827
Test 2	30.3959	3.792
Test 3	30.4636	3.867
Average	30.5082	3.829

The speedup from parallelising over serial for this dataset is 7.968 times as fast.

This set had 200,000 Data points with 4 Features classified into 4 clusters. There is twice as much work to do, as can be seen from the serial implementation taking just under twice as long on average, and even then, only by 400ms. The parallel implementation however takes less than twice as long, with an increased speed up of 1.845 times more of the serial processing speed. I believe this is due to the extra time the serial implementation would have to spend managing such a large dataset in memory.

SET 6	Serial	Parallel
Test 1	24.8978	3.184
Test 2	25.3175	3.287
Test 3	25.5069	3.450
Average	25.2407	3.307

The speedup for this dataset is 7.6325 which is similar to the speedup from the previous set. This may be because of the near identical level of data, 200,000 lots of sets of 4 floating points numbers and 100,000 lots of sets of 8 floating points numbers, being processed in an identical way.

Again, this could most likely be due to the large amount of memory required to house 800,000 floating point numbers.

SET 7	Serial	Parallel
Test 1	29.8862	3.115
Test 2	29.5609	3.149
Test 3	29.9063	3.270
Average	29.7845	3.178

The speedup for this dataset is 9.372 times.

This far greater speedup is due to there being the same 100,000 data points and 4 features but instead having the data be classified into twice as many clusters. This is perfect for the parallel implementation as the level of data remains the same, but the processing time is increased by a factor of 2. The amount of time dispy has to take to transfer the data to the nodes and process the results is the exact same. The Equation (`MainTime + NodeTime`) can be used to describe how long the cluster takes to complete its calculations. MainTime being how long the client board spends preparing, send, receiving and unpacking data, and NodeTimes used to describe how long the server nodes spend performing the calculations. For this dataset, one part of the total time equation is identical. For the serial implementation, the data is not split this way and no preparations need to be made, and as such the whole process takes twice as long, rather than just a portion.

SET 8	Serial	Parallel
Test 1	96.5901	7.435
Test 2	98.3196	7.553
Test 3	99.8832	7.737
Average	98.2643	7.575

For this dataset, all key values are doubled. 200,000 Data points with 8 features classified into 8 clusters. The speedup for this test is 12.972. The longer the mapping function takes to run, the less impact the MainTime variable in the time equation becomes, hence the dramatic increase in the speedup. Also, this dataset benefits both from the probable increased efficiency of memory management from the large dataset, but also the increase in NodeTime over MainTime like the previous dataset.

5.2) Taylor Series Expansion (Arctan)

The main variable for comparison with the Taylor Series is the precision to which the approximation is calculated. The more of the series that is calculated, the closer to the true value can be found and the longer the calculation will take.

For testing a range of values between -1 and 1 (0 non inclusive) were chosen to be the value of X in f(x) = arctan(x). the reason -1, 0 and 1 were not chosen is because these values provide extraordinary results in the Taylor series expansion for arctan and would require a different function.

6 10-digit precision floating point values were chosen at random as follows: -0.9999999999, 0.9999999999, 0.0000000001, 0.7853194857, -0.8200000000 and 0.4792378463.

It should be noted that values close to -1, 0 and 1 are much more difficult to approximate. These values are included to show the benefit of having a multitude of elements from the Taylor Series to increase accuracy.

Each element in the Taylor Series expansion moves the approximation a step closer to its convergence with the real value of arctan(x). Each of these values was tested with 4 different degrees of approximation: 32 steps - two calculations per node, 1024 steps – 64 calculations, 16384 steps – 1024 calculations and 131072 – 8192 calculations per node.

Timing Results (Serial):

Values \ Steps	32	1024	16384	131072
-0.9999999999	0.000542	0.005980	0.100361	0.945741
0.9999999999	0.000558	0.007718	0.084020	0.891919
0.0000000001	0.000565	0.005733	0.084777	0.901012
0.7853194857	0.000538	0.005721	0.085244	0.896168
-0.8200000000	0.000518	0.005952	0.091560	0.949272
0.4792378463	0.000519	0.005716	0.084808	0.905683
Average	0.000540	0.006140	0.088460	0.914970

Timing Results (Parallel):

Values \ Steps	32	1024	16384	131072
-0.9999999999	0.714	0.752	0.707	0.907
0.9999999999	0.621	0.608	0.601	0.703
0.0000000001	0.622	0.622	0.635	0.708
0.7853194857	0.624	0.660	0.601	0.704
-0.8200000000	0.632	0.639	0.612	0.690
0.4792378463	0.629	0.638	0.611	0.697
Average	0.640	0.653	0.626	0.734

As can be seen from the results from the K Means experiments, the more work there is to be done, the better the parallel cluster fares. Very small numbers of calculations provide a horrible slowdown. The serial implementation increases linearly with the amount of calculations needed to be done as is expected, whereas the parallel cluster has a constant value and a linear increase that is far smaller than the serial implementations.

In the first 3 precision trials, the time taken to do the calculation is lost in the variability of network latency. Its only when a truly significant number of calculations must occur that the processing time becomes obvious. The parallel implementation saw an increase of approximately 110ms from the 3rd to 4th precision step, whereas the serial implementation saw an increase of 826ms.

These values could be used to provide a more accurate speedup value as it does not include the constant value of approximately 620-630ms. This is unchanging as the level of data being transmitted remains constant: 2 indicies and a floating point number. Using this the speedup can be calculated as 7.509 times as fast. This is true for the function calculation alone, however, the constant transmission delay must be considered when evaluating the use of this function. Will the time saved from the parallel function outweigh the cost of using the parallel function in the first place?

Assuming each calculation takes the exact same amount of time, and that the delay of using the parallel function is constant, which is a rather large and unwieldy assumption to make, the exact number of calculations needed to be completed for the parallel function to take a shorter amount of time can be calculated by

`SerialTime*ParallelConstant*Number`

For the 4th precision step, this comes out to 75,554. So approximately 75,500 calculations must be undertaken for the parallel function to provide any measure of positive speedup at all.

Approximation Accuracy (% Similarity):

Calculated as (`100*|real-approximation|/real`) with 100% being identical to 15 decimal places. The real value was obtained from Pythons Math module using the supplied arctan function.

The Taylor Series expansion of arctan is deterministic and as such both the Serial and Parallel implementations return the same value.

Values \ Steps	32	1024	16384	131072
-0.9999999999	-22.322%	96.656%	99.792%	99.974%
0.9999999999	-22.322%	96.656%	99.792%	99.974%
0.0000000001	100%	100%	100%	100%
0.7853194857	70.455%	99.9…94%	99.9…91%	99.9…91%
-0.8200000000	65.784%	99.9…98698%	99.9…9%	99.9…9%
0.4792378463	91.598	100%	100%	100%

The values that have been shortened reached 15 decimal places and are only shortened to facilitate easy reading.

It is interesting to note the identical results for the first 2 numbers. The absolute value of arctan function is symmetrical around x = 0.

Realistically, a suitable level of precision can be found from 1024 units of precision for number ~-0.8 - ~0.8. 16384 units of precision seems suitable for all other numbers that are closer to the limit than a floating point value could represent.

It seems the closer the value is to the limits of the function, -1 and 1, the higher the precision needs to be to obtain an accurate approximation, with even 131,072 elements from the Taylor Series still not quite giving perfect floating point accuracy.

5.3) Binary Search

The binary search function was performed on a set of datasets consisting of integers between 0 and 4000. The Numpy random.randint function was used, which provides a pseudo random distribution of numbers. This dataset was randomized each time and one number chosen at random from the dataset to search for. The search function was run on a dataset of 1000, 10,000, 100,000 and 1,000,000 to gain a measure of its effectiveness.

All times recorded are in seconds.

SERIAL	Test 1	Test 2	Test 3	Average
1000	0.010374010	0.010508489	0.010700209	0.010528
10,000	0.111309271	0.104826615	0.108468750	0.108202
100,000	1.111506614	1.066984270	1.034398385	1.070963
1,000,000	10.57015140	10.44323036	10.94189312	10.65176
PARALLEL	Test 1	Test 2	Test 3	Average
1000	0.736	0.738	0.753	0.742333
10,000	0.790	0.786	0.757	0.777667
100,000	0.799	0.904	0.880	0.861
1,000,000	1.565	1.603	1.604	1.590667

As seem before with the Taylor Series expansion the meeting point for serial in parallel in time taken seems to be slightly under 100000 computations. The actual processing needing to be done by each node is so small that tens of thousands are needed before the cluster is made viable.

Each node is ideally performing 1/16th of the raw processing of the serial computation, though this an assumption due to the other server side processing that keeps the node online. For 1000, this is an average of less than 6.5ms. Taking that away from the average time taken for 1000 for parallel, this leaves a constant of 0.741s. Taking this figure away will show us the theoretical maximum speedup possible from the cluster. As the number of datapoints needing to be processed tends to infinite the proportion of time spent as the constant will tend to 0. This means for 1,000,000 data pieces the cluster should have spent 0.850 seconds of it actually performing the computation. Using this and the time for the serial code, a theoretical speed up of 12.537 is possible, whereas realistically it only achieved a speedup of 6.696. These numbers will draw closer as the amount of data pieces being processed increases.

6. Conclusion

The results collected in this investigation are strikingly clear. A small cluster of low power devices can be used to provide a remarkable speed up to serial code, an average of 7-8 times as fast, for a relatively low coding cost. This is unlikely to be ground-breaking news to many developers who themselves are taking advantage of the power a cluster can provide to a coding solution (Pfister 1998). However, a traditional cluster is usually built either out of spare workstations cobbled together to scrape as much performance as possible out of them, or a highly specialised, dedicated set of hardware that is built with a single purpose in mind.

This project provides an alternative path of standard low budget, low power, off the shelf equipment ready to be used in a wide variety of projects. As has been demonstrated, a cluster such as the "Lasagna" can provide benefits for incredibly simple code, such as a binary search, and much more complex code, in the form of machine learning. More boards can be added to increase the benefits available, however, as discovered when testing the 3rd K Means dataset, high data volumes can be bottlenecked by their transmission speed.

A secondary use for this project could be in extending the life of old serial by refactoring it for use in a cluster. If the main computation can be performed independently of its data requirements and extracted from the main workflow, it can then be performed in parallel providing a large performance boost to help slow code remain useable for far longer.

The results obtained do make one thing abundantly clear however, this cluster needs to have a suitably large task to perform in order for its time costs to be outweighed. Low complexity individual tasks such as basic mathematics tasks take such a short time to compute that an excessive amount are needed to for this architecture to provide a benefit. These kinds of tasks are better suited to conventional GPGPU's whose architecture is perfect to support a multitude of small calculations.

With this specific hardware, a given computation time of approximately 600ms seems to be the absolute bare minimum that the "Project Lasagna" can provide any benefit. This entirely rules out any sort of real time operations such as on the fly machine learning, optical recognition, video processing or graphical work that standard parallel hardware would usually be able to cope with.

For operations longer and more complex than that however, the hardware available can provide a tangible boost to performance and long term savings in upkeep and electricity costs that more than offsets its moderate investment cost. A conventional domestic GPGPU can cost upwards of £1000 with a stressed TDP of over 250W. All in this cluster cost less than £300 and came ready to use out of the box whereas a conventional GPGPU still needs a housing computer. This cluster also draws less than 40W when under stress, under 6 times the power of a GPGPU without the computer its connected to.

This project outlines that low power microprocessors can most certainly be used to create a budget pseudo multiprocessor that can effectively run complex parallel code.

7. Legal, Ethical, and Professional Considerations

The dispy framework library uses the Apache 2.0 usage licence. This allows “a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form”.

The Raspberry Pi board is sold without license and is not subject to any usage terms.

All novel code displayed in this report is to be used under the Apache 2.0 Usage License as described above in accordance with the usage of the dispy licence.

APPENDIX A – BIBLIOGRAPHY

Smotherman, M., (Revised 2019, now defunct) Intel Hyper-Threading Technology, viewed 14 April 2020 <www.intel.com/cd/00/00/01/77/17705_htt_user_guide.pdf>

Crowell, T., (2010) Architecting Solution for the Manycore Future, viewed 13 April 2020 <https://www.slideshare.net/Talbott/architecting-solutions-for-the-manycore-future>

Upton, E., (2015) Raspberry Pi 2 on sale now at $35, viewed 4 February 2020 <https://www.raspberrypi.org/blog/raspberry-pi-2-on-sale/>

Hull, G., (December 1987). "LIFE". Amazing Computing. 2 (12): 81–84.

Unknown Author, (Revised 2018) Common-Shader Core, viewed 15 April 2020 <https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-common-core>

Zeller, C., (2011) CUDA C/C++ Basics, viewed 22 February

<https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf>

Schuh, A. (2010) MCUDA Download Page, viewed 13 April 2020 <http://impact.crhc.illinois.edu/mcuda.aspx>

Moon, H., (2016) Parameters That Affect Parallel Processing For Computational Electromagnetic Simulation Codes On High Performance Computing Clusters

Pfister, G., (1998). In Search of Clusters (2nd ed.). Upper Saddle River, NJ: Prentice Hall PTR. p. 36. ISBN 978-0-13-899709-0.

Author Unknown (Date Unknown) NVIDIA Quadro RTX 8000, viewed 14 April 2020 <https://www.nvidia.com/en-gb/design-visualization/quadro/rtx-8000/>

Raspberry Pi Ltd, (2016) Raspberry Pi Compute Module Datasheet

Spiegel, M., (1968). Mathematical Handbook of Formulas and Tables. 20.29

DEVICE REFERENCES

http://beagleboard.org/black

https://www.raspberrypi.org/products/raspberry-pi-zero-w/

https://store.arduino.cc/arduino-yun

https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/