Macrothink Institute™

Network Protocols and Algorithms
ISSN 1943-3581
2016, Vol. 8, No. 4

A Scalable Cloud Based on Commodity Hardware

Saibal K. Ghosh and Dharma P. Agrawal

Dept. of Electrical Engineering and Computing Systems

University of Cincinnati, 812 Rhodes Hall, Cincinnati, Ohio 45221. United States

E-mail: ghoshsl@mail.uc.edu, agrawadp@ucmail.uc.edu

Received: October 12, 2016 Accepted: December 28, 2016 Published: December 31, 2016

DOI: 10.5296/npa.v8i4.10291 URL: http://dx.doi.org/10.5296/npa.v8i4.10291

Abstract

The recent explosion in the speed and connectivity of the Internet has opened up the possibility of millions and possibly billions of devices connected together. Combined with the development of small, low power devices, new paradigms in the field of computing have opened up. Traditional passive electronic devices now have rudimentary computing capabilities. The resulting Internet of Things (IoT), comprised of smart interconnected devices is improving our ability to gather ambient information and make informed decisions that directly benefit humanity. However, the ubiquity of these devices also presents an interesting scenario wherein the devices can perform limited general-purpose computations when they are not performing their primary functions. A computational task divided into a large number of smaller, micro tasks, each of which take only a few CPU cycles to complete. By distributing these tasks over a large number of devices, we can achieve a substantial amount of computation with seemingly modest devices. In this work, we explore a mechanism to enable such massively parallel computations in low powered commodity hardware devices through fine-grained task parallelism.

Keywords: Cloud Computing, Internet of Things (IoT), Machine-to-Machine Communications (M2M), Resource Allocation, Fine grained Task Parallelism, Virtual Machines, Geographic localization.

1. Introduction

The explosive growth of the Internet in recent years has given rise to a plethora of applications. These applications benefit from running on computing devices that are usually connected and being able to exchange information through a myriad of wired and wireless connections. Traditional devices that had a singular purpose now have computing and networking capabilities. This has given rise to the paradigm of the Internet of Things wherein everyday mundane objects like toasters and refrigerators are becoming smart thanks to their embedded processing power and Internet connectivity [1] [2] [3]. Given the sheer numbers of these devices, a large cluster of such low powered devices can deliver substantial processing power [4]. Traditional cluster computing relies on a fixed set of nodes, connected through proprietary interconnects and running a highly customized operating system. This has generally made cluster computing a forte of universities and other research institutes with large budgets. A large number of low powered devices are able to work together on a particular computational problem and possibly deliver results faster than a single conventional high performance computer. With the prevalence of network connectivity, today's clusters can have hundreds or thousands of machines and can therefore target computations of increasing complexity. Computers from all across the world can be connected and take part in large-scale computations [5] [6]. However, while processor speeds have steadily increased, the network bandwidth varies considerably in various parts of the world and is a significant hindrance to the growth and popularity of collaborative computing.

Traditional grid computing initiatives have tried to circumvent the network connectivity problem by eliminating peer-to-peer communication altogether [7] [8]. A central server splits the task into chunks and assigns them a unique identifier. Nodes taking part in the computation connect to this central server, download a chunk and start their computation. Independent computations run on each chunk. Once a particular node finishes its computation, it uploads the results back to the server. Once all the chunks have finished computation, the server assembles the computed chunks to get the result [9]. While this alleviates the problem of network latency, it suffers from a number of problems. A client only contacts the server to download new chunks or to report completed tasks. The server has no way of knowing the progress of the chunks in each client. Thus, a computational task might have to wait for one slow client or risk delay due to network failure for a single client. Servers generally set a timeout for all chunks in order to take care of this problem. If the client does not contact the server with the completed work before the timeout, the server considers the chunk lost and sends out another copy of the chunk. Therefore, this eliminates the problem of one node slowing down the whole computation at the cost of adding a finite amount of delay for every chunk.

If, a mechanism were available such that the nodes are able to communicate in real time with each other and the server, it would greatly improve on the aforementioned problems. All the nodes would be able to work more efficiently since they would be working on smaller data sets. Continuous exchange of data between the peers would ensure identification of slow performing nodes and less demanding tasks to the identified nodes. The system would also recover from failures early and be able to roll back quickly since the server is aware of the

current progress of the task in all the different nodes. While, the current global network latencies are not comparable to a dedicated computing cluster, we can exploit the current global network behavior in order to improve on the throughput.

In this work, we evaluate mechanisms to enable distributed computing using low powered commodity hardware by reducing network latencies in a globally distributed computational grid. Based on the nature of computation, the resources dedicated for a particular task being executed change thereby making our system elastic. This elasticity makes it possible to take advantage of increased computational power when available and rollback and resume computations when the available computation power dwindles. We also extend our study to evaluate a method to improve the computation time by bringing together devices through geographic and network localizations.

The paper is structured as follows: Section 2 discusses the background about parallel computation and the differences between theoretical and achievable speedups. We also discuss the relationship between network latency and geographic proximity. We then extend these ideas to show how fine-grained parallelism over the network speeds up computations. Section 3 describes our geographically distributed grid and the experimental setup for the simulations. Section 4 discusses the results and the speedups achieved from our mechanism. Section 5 discusses our future work and concludes the paper.

2. Background and Related Work

The availability of a multitude of internet-connected devices opens up the possibility of performing massively parallel computations. Small low-powered commodity devices can be grouped together to build vastly powerful computing engines. These commodity devices range from Network Attached Storage (NAS) devices to intrusion detection and monitoring devices to refrigerators and toasters of tomorrow. While these devices often have relatively weak CPUs, since they are only required to perform a single task, they generally have long idle periods and thereby spare CPU cycles. These CPU cycles can be utilized to perform small computations which when performed in parallel can produce a substantial computation throughput. This is an idea that we will explore in this work. We envisage an open system that harnesses the spare CPU cycles of small low powered devices to provide a significant computational engine.

However, we should realize that performance improvements in computations are not always directly proportional to the level of parallelism achieved. Again, many performance improvements in traditional computer programming relates directly to temporal and spatial localizations. High performance computer code relies on these localizations to extract the best possible performance from the hardware it is running on. Since our approach involves splitting a large computational workload into multiple smaller workloads, we have to ensure that we utilize the location affinities. As these smaller workloads divided across multiple devices working in varied geographic locations, we take into account the geographic proximity of these devices while splitting and assigning the workloads. In this section, we

would give a brief background on the amount of actual performance improvements possible through parallelism. We then show how we have improved the performance through fine-grained parallelism and geographic localization.

2.1 Theoretical and achievable gains in parallel computation

The nature of computer programs dictates that the performance gains arising from executing a program in parallel are not always linear. This is because introducing concurrency only affects parts of a program. Therefore, the actual gains in performance are usually lower than expected from a naive ballpark analysis. Consider a program that performs multiple database lookups over a network. Introducing concurrent database access into the program would only improve the performance by a percentage related to the percentage of time that is spent for the database access over the network. If the fraction of the overall execution time spent by the program accessing the database is low, we might not actually see any appreciable improvements in performance. The performance gain in a computing environment is limited by the fraction of the time that the computation can actually speed up. For any computation, there would always be certain areas that cannot execute in parallel. Therefore, the theoretical speedup achieved from parallelism is always the sum of the time that the computation runs in parallel and the time it has to run in sequential mode as shown in Equation 1.

$$Execution\ time_{new} = \frac{Execution\ time_{affected}}{Improvement} + Execution\ time_{unaffected} \quad (1)$$

The relationship in Equation 1 is Amdahl's Law [10]. Therefore, the new execution time is the sum of the execution time of the parts of the program that are unaffected by the introduced concurrency and the improvement on the execution times in the parts of the program that could be successfully executed in parallel.

Therefore, the speedups achievable in a parallel computing scenario is computed as a function of the percentage of execution time that can be made parallel and the number of processors as shown in Equation 2:

$$Speedup = \frac{1}{1 - P + \frac{P}{N}} \quad (2)$$

Where, P is the percent of execution time that is parallel and N is the number of processors. One way to improve the speedup is to increase the number of processors such that a larger amount of the computation runs in parallel. Now, as N approaches1, the achievable speedup only becomes a function of the percentage of execution that can be run in parallel as shown in Equation 3:

$$Speedup = \frac{1}{1 - P} \quad (3)$$

This is because as the number of processors approaches ∞, the ratio of the percentage of execution time that can be made parallel and the number of processors approaches 0 as shown in Equation 4:

$$\lim_{N\to\infty} \frac{P}{N} = 0 \tag{4}$$

Therefore, merely increasing the level of concurrency in computation does not necessarily improve its throughput. In order to achieve an optimum level of throughput a number of factors require fine-tuning to work in tandem. A combination of fine-grained parallelism and efficient techniques to reduce network latency would be essential in improving the overall throughput. We explore these approaches in this work through network localization and our metrics for distributing workloads to the most eligible nodes.

2.2 Network latency and Geographic localization

Studies on localization in peer-to-peer systems have mainly concentrated on file sharing systems aiming to reduce inter-ISP traffic since that is the major cause of bottlenecks on the Internet. Le Blond et al. have conducted experiments with a large number of peers and have shown that high localization is able to reduce inter-ISP traffic by two orders of magnitude [11]. Karigiannis et al. have shown that even simple mechanisms to enable locality awareness in peer to peer networks can substantially improve the network performance and approximate it to a perfect world-wide caching infrastructure [12] [13]. Implementing these changes require the peer discovery mechanisms to be modified such that neighboring peers are prioritized over others in the swarm [14] [15]. While these studies focus on improving throughput in bulk data transfers in file sharing systems, our study is towards fine-grained task parallelism on the Internet and involving limited data transfers with more emphasis on latency minimization.

In any distributed system, the key to performance is minimization of latency. This is especially important in a geographically distributed system wherein the network parameters that govern the actual latencies are difficult to control, unlike in a conventional computational grid. Geographic distances are directly proportional to the network latencies since messages have to traverse more autonomous system boundaries. Although certain autonomous systems have direct connections between them, the Internet as a whole behaves quite similarly to national and geographic boundaries. Therefore, in order to make the most effective usage of the computing nodes, we need to detect and group together nodes that have minimal latencies, possibly in the same autonomous system.

We therefore attempt to identify the nodes that are geographically close and group them. However, as mentioned previously, sometimes seemingly distant autonomous systems might have lower latencies owing to the peering policies between the network providers and multiple network connections between two autonomous systems. Moreover, the peering policies between two autonomous systems change based on the nature of the current Internet traffic between them. For example, when a popular sporting event such as the Olympics streams simultaneously across the world, network providers enter into multiple peering

agreements with the providers in the host country in order to route the temporary increase in network traffic efficiently for the duration of the event. These agreements might call for redundant and/or high capacity, low latency links between the network providers. Hence, the nature of Internet traffic is always in a state of flux and the true latency depends on a number of factors. It is almost impossible to come up with fixed numbers for the traffic quality on the Internet. We therefore take into account the real-time network latencies between the various autonomous systems and continuously perform a weighted average over all these factors to come up with a metric for the actual proximity between two devices. We explain this process in more detail in Section 4. As autonomous systems have unique identifiers for all their component nodes, we can use this to our advantage.

We would now attempt to establish a relationship between the geographic proximity of two computing nodes on the internet and their perceived network latencies. The proximity of two points on the earth is determined using the Haversine formula [16].

Consider, two computing nodes X_1 and X_2 and their corresponding latitude and longitude as $X_1(lat_1, lon_1)$ and $X_2(lat_2, lon_2)$. The actual geographic distance between the nodes is the great-circle distance, which is the shortest distance between two points on the surface of the Earth as measured along the surface of the Earth. We can get an estimate of the great-circle distance between the nodes as a function of the Earth's radius and the geographic coordinates (latitude and longitude) for a point on the Earth as shown in Equation 5:

$$d = R \bullet c \tag{5}$$

Where, R is the Earth's mean radius of 6,373 km (3961 mi) and the parameter c is in Equation 6:

$$c = 2 \bullet \arctan 2(\sqrt{a}, \sqrt{1-a}) \tag{6}$$

The parameter "*a*" is determined from the geographic coordinates of the two points on the surface of the Earth as shown in Equation 7:

$$a = \sin^2 \frac{lat_1 \sim lat_2}{2} + \cos(lat_1) \bullet \cos(lat_2) \bullet \sin^2 \frac{lon_1 \sim lon_2}{2} \tag{7}$$

The parameter arctan2(y, x) is evaluated based on the values of the other parameters as shown in Equation 8. Computing this value for two points on the surface of the Earth gives us an estimate of the great circle distance between them and therefore their proximity.

$$arctan2(y,x)=\begin{cases} arctan\frac{y}{x}, & \text{when } x>0 \\ arctan\frac{y}{x}+\pi, & \text{when } y\geq 0 \text{ and } x\leq 0 \\ arctan\frac{y}{x}-\pi, & \text{when } y<0 \text{ and } x<0 \\ +\frac{\pi}{2}, & \text{when } y>0 \text{ and } x=0 \\ -\frac{\pi}{2}, & \text{when } y<0 \text{ and } x=0 \\ undefined, & \text{when } y=0 \text{ and } x=0 \end{cases} \quad (8)$$

2.3 Fine grained parallelism and computational workloads

Given the fact that our system targets smaller, modest computing devices, in order to perform an effective computation, run, they should effectively be computing a thin slice of the total computation since almost of them would be constrained by the CPU and the available physical memory. Our approach is to determine the individual fine-grained computations that comprise the overall task. While the overall task might have been, too resource intensive for any of these devices to handle, these much smaller sub tasks can effectively run on these devices with an acceptable turnaround time. These fine-grained tasks comprise of only a few machine instructions that can run repeatedly and efficiently on these individual resource constrained devices.

In order to evaluate the performance of our system we have used general-purpose computing tasks distributed across the computing nodes. Signal processing applications use the Discrete Fourier Transform (DFT) to convert signals between the time and frequency domains. We will show later how the DFT is suited for fine grained parallelism and how we have exploited this property to run massively parallel DFT computations with large input arrays. The DFT, *X(k)* of a finite signal of discrete length *x(n)* is [20]:

$$X(k)=\sum_{n=0}^{N-1} x_n \bullet e^{\frac{-i2\pi kn}{N}} \quad (9)$$

The original signal for the DFT computation is [20]:

$$x(n)=\sum_{k=0}^{N-1} X_k \bullet e^{\frac{i2\pi kn}{N}} \quad (10)$$

Since we have assumed a periodic signal with a period that is equal to the length of the signal sequence *x(n)*, the DFT has a finite boundary. Now, in order to compute the DFT of a finite length signal of length N, we need to perform $4N^2$ multiplications and N(4N-1) additions, which give us a time complexity of $O(N^2)$. Therefore, in order to improve the performance, we need to run the DFT computations in parallel. Parallel Fast Fourier

Transform computations take advantage of multiple processing nodes that can run computations independently of each other to achieve an improved throughput. Cooley and Tukey in their seminal paper have shown that the Discrete Fourier Transformation computation decomposes into two smaller transforms as in Equation 11 [17] [18]:

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n \bullet e^{\frac{-i2\pi kn}{N}} \\ &= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} \bullet e^{\frac{-i2\pi k(2m)}{N}} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} \bullet e^{\frac{-i2\pi k(2m+1)}{N}} \\ &= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} \bullet e^{\frac{-i2\pi km}{\frac{N}{2}}} + e^{\frac{-i2\pi k}{N}} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} \bullet e^{\frac{-i2\pi km}{\frac{N}{2}}} \end{aligned} \tag{11}$$

The decomposition of this DFT breaks down the transform into two smaller transforms based on the odd and even numbered values. These two computations can be performed independently of each other and thereby improve the computation time. Independent computations are important since these can run on separate devices with minimum communication requirements between the computing nodes.

Now, using mathematics we can further reduce the computations for the X_{N+k} term as shown in Equation 12:

$$\begin{aligned} X_{N+k} &= \sum_{n=0}^{N-1} x_n \bullet e^{\frac{-i2\pi(N+k)}{\frac{n}{N}}} \\ &= \sum_{n=0}^{N-1} x_n \bullet e^{-i2\pi n} \bullet e^{\frac{-i2\pi kn}{N}} \\ &= \sum_{n=0}^{n-1} x_n \bullet e^{\frac{-i2\pi kn}{N}} \end{aligned} \tag{12}$$

Now, the $e^{i2\pi n}$ term in Equation 12 evaluates to 1 for all values of n. Utilizing this property, we can get two interesting relations for the X_{N+k} and the X_{k+i} terms as shown in Equations 13 and 14:

$$X_{N+k} = X_k \tag{13}$$

And,

$$X_{k+i} \bullet N = X_k \tag{14}$$

This relation is the symmetry property of Fast Fourier Transforms. The values of *k* and *n* are in the ranges as shown in Equations 15 and 16:

$$0 \le k < N \tag{15}$$

$$0 \le n < M \equiv \frac{N}{2} \tag{16}$$

Utilizing this property eliminates half of the computations and we get a complexity of *O(N/2 log N)*. This approach of breaking down the input into its constituent parts and computing the values of the resulting parts is termed as decimation and works as long as the input is an even integer. An upper bound to the number of decimations can be carried out on an input array exists since the resultant input arrays become so small that no further computational benefit can be achieved. However, with each step, the potential of implementing parallel computation increases since all the constituent parts of the original FFT runs independently.

3. Experimental Setup

In keeping with the tenets of Moore's Law, we have seen transistor density in processors double every eighteen months over the last few decades. However, the recent years have seen chip-manufacturing processes reach theoretical bottlenecks in clock speeds and the transistor densities on the chips. The response has been a move towards multicore processing. The initial approaches involved multiple dedicated CPUs running on a single host motherboard with a common shared memory between them. Recent designs involve multiple cores on the same CPU. This approach increases the amount of high-speed memory shared between the cores and allows for greater throughput. Thus, while clock speeds have remained constant, processing power has continued with a steady increase with the increase in the number of cores on a single chip. However, this increase in processing power comes with an increase in the power requirements as well. The Internet of Things (IoT) is generally composed of small low powered devices that run on batteries. Recent advances in the development of power efficient chips that can dynamically modify their clock speeds have led to their prevalence in the IoT devices. On the other hand, developments in communication technologies have led to better utilization of the network bandwidth and higher network speeds. We envisage the IoT scenario of the future wherein we have a large number of small, power-efficient computing devices connected over high-speed data links. Accordingly, our experimental setup uses modest CPUs connected over high bandwidth network links. We take advantage of this high connectivity to run our simulations with fine-grained parallelism**.**

In order to run our experiment, we setup small servers in four countries (the United States, Canada, the United Kingdom and Germany) running a minimal version of Debian Linux. These were inexpensive bare metal micro servers with modest amounts of RAM ($\approx$ 64 Mb), moderate SSD storage ($\approx$ 5 GB) and a single core CPU running at around 2GHz. However, all servers connect to the Internet with high-speed data links (guaranteed $\geq$ 100 Mbit/sec). These specifications mimic the computational capabilities of devices that we envisage to form the core of IoT devices. The high-speed links in these servers were crucial to our experiment, as we had to pass many messages between these servers, which were often located in different autonomous systems.

3.1 Proximity matrix computation

In order to determine the geographical proximity between two nodes we use the great circle distance as mentioned in Section 3. We first estimate the latitude and longitude of the physical location of servers from their IP address. We use the Maxmind IP database service for the same [19]. We compute the great circle distances between each pair of nodes from their associated latitude and longitude. The proximity matrix contains weighted values that are a determined from the physical distance and the latencies between each pair of nodes. Therefore, the proximity matrix reflects the actual proximity of the nodes from the network and computation point of view. Smaller values in the table indicate that the nodes are closer while larger values indicate that the nodes are further apart. Note that two nodes might be close in the matrix even if they are geographically further apart although we have found this to be extremely rare from our studies. Nodes that are close together are prime candidates to take part in the same computation. Normally, we compute the proximity matrix at the following times:

- Start of computation
- When a new computing node comes online
- Every fifteen minutes for the total duration of the task

At the start of each computation, we allocate a global timer at the UC lab server. This timer keeps track of the progress of the computation and initiates the proximity matrix computation every 15 minutes. We also start a daemon process to monitor if a new node comes online. Once a node comes online, it determines its closest node coordinator by measuring the latency and publishes its latency measurements. We use these measurements to update the proximity matrix continuously. Nodes that go offline would not be able to publish their latency measurements and would require a latency measurement at the end of the next fifteen minutes. Fig. 1 illustrates the process of computing the proximity matrix as a flow diagram.

In addition, each node continuously monitors the latencies to all the nodes that it is dependent on for its computation. In case of a variation beyond a predefined acceptable range, the node may raise a request to perform a re-computation of the proximity matrix for that region. At the start of a computation run, all the participating nodes are aware of the presence of the other participating nodes. The proximity matrix computation at the start of the simulation serves to notify each node of the presence of the other participating nodes and the network map. A new node coming online increases the computational capability of the system, as a whole but also requires a computing the network map again. The rationale behind computing the proximity matrix every fifteen minutes is that the network conditions are always in a state of flux. While we understand that continuous computation of the proximity matrix would give an accurate picture of the state of the network, we also realize that computing the network map requires some computing resources. Thus, our approach of generating the proximity matrix every fifteen minutes strikes a balance between the computing resources needed and the accuracy of the network map. Table 1 shows a snapshot

of the proximity matrix. All nodes participating in the computation keep track of the distance matrix and updates values at specified times. The proximity matrix updates with the heartbeat messages exchanged between the nodes and the local coordinators is at periodic intervals as explained in the following section. Any node in the system can perform a proximity matrix lookup and determine its closest neighbors.

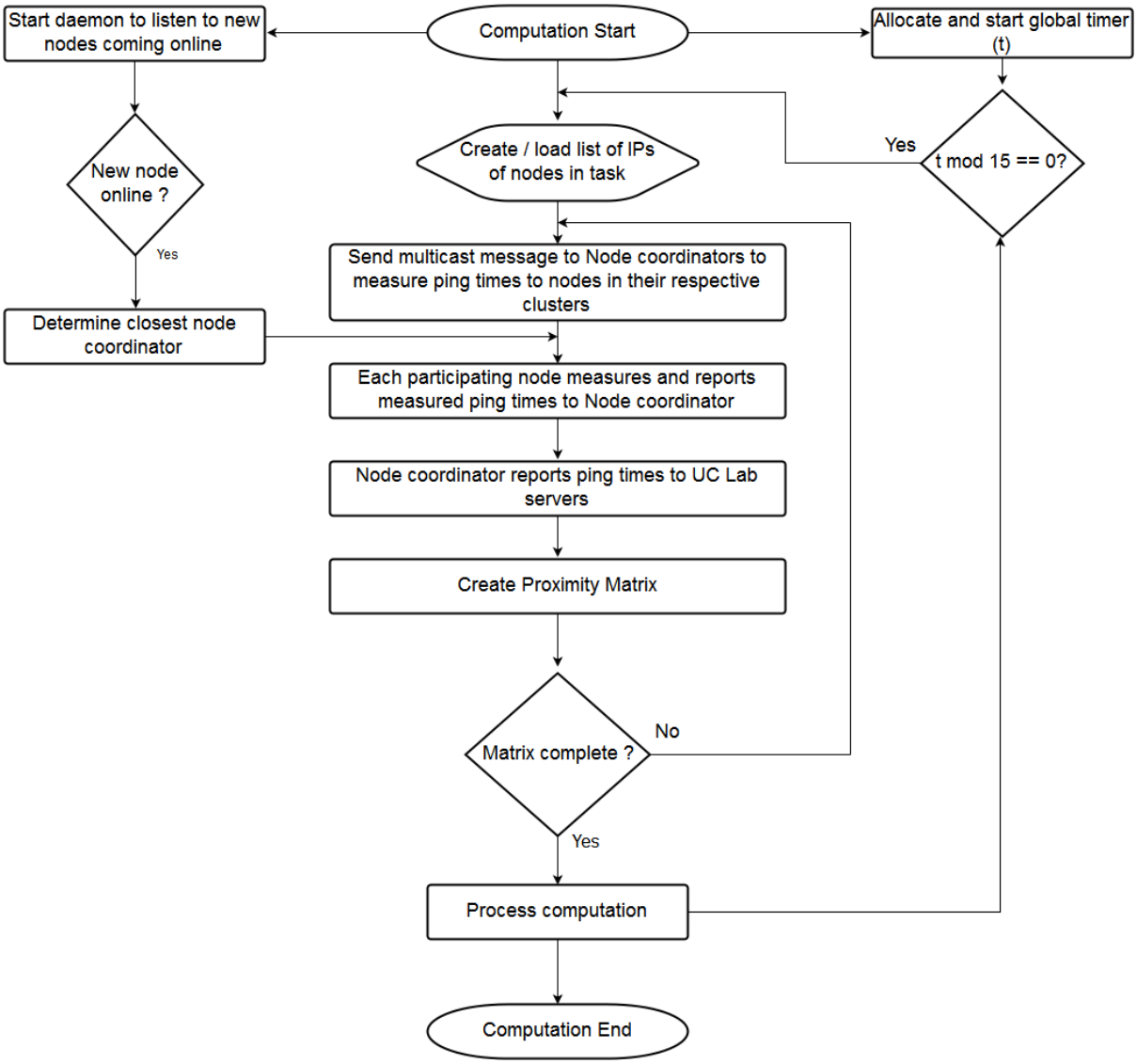

Figure 1. Flow diagram showing the Proximity Matrix computation

3.2 Node hierarchy and task distribution

Each running computation on our system is comprised of a number of nodes running a part of the overall computation. These nodes form a tree with the University of Cincinnati (UC) Lab servers forming the root of the tree. The UC Lab servers are responsible for splitting the workload into multiple fine-grained smaller workloads sent to each participating node. Each node then spawns a process that performs the actual computation. A single node might have two processes performing computations that are part of two different workloads.

Table 1. Proximity matrix snapshot for the geographically distributed servers

	US_1	US_2	US_3	US_4	CA_1	CA_2	CA_3	CA_4	GB_1	GB_2	GB_3	GB_4	DE_1	DE_2	DE_3	DE_4
US_1	0.0	4.7	2.1	2.8	2.2	5.3	2.8	4.7	3.7	4.4	3.6	6.2	8.4	8.7	6.8	7.2
US_2	4.1	0.0	1.8	3.7	4.1	4.8	3.8	5.3	4.6	4.1	3.8	6.6	8.1	8.9	5.8	6.9
US_3	2.3	2.1	0.0	3.4	3.8	4.3	4.1	4.4	5.1	4.8	3.4	6.1	7.8	8.2	5.2	7.6
US_4	3.1	3.4	3.7	0.0	3.6	4.4	4.3	5.1	4.7	5.2	3.2	6.8	8.4	8.2	5.5	7.8
CA_1	1.6	4.3	3.6	3.8	0.0	2.1	2.8	3.4	4.7	4.4	5.9	6.0	6.4	6.7	7.2	8.3
CA_2	5.1	4.5	4.1	4.6	1.8	0.0	3.8	4.1	5.3	5.7	4.8	5.6	6.8	7.7	6.9	8.1
CA_3	2.4	5.1	3.8	3.9	2.6	3.9	0.0	5.1	5.8	4.9	5.1	6.2	7.1	7.3	7.1	7.9
CA_4	5.9	5.5	4.7	5.3	3.1	3.6	4.9	0.0	6.2	5.8	5.3	6.8	7.6	7.1	6.8	8.3
GB_1	3.7	4.4	5.3	4.6	4.1	5.5	5.5	6.4	0.0	2.7	3.4	2.4	5.7	5.3	6.1	5.8
GB_2	4.1	3.8	5.1	5.3	5.2	6.2	4.3	6.9	2.3	0.0	4.1	3.3	4.7	5.8	5.8	6.3
GB_3	4.7	4.1	3.8	3.3	5.7	5.1	4.8	6.7	2.6	3.9	0.0	1.6	3.9	4.4	4.8	5.9
GB_4	6.6	6.2	5.7	7.1	6.3	6.3	6.1	7.2	2.8	3.6	1.4	0.0	4.3	5.1	5.8	6.3
DE_1	8.1	7.8	8.3	8.1	5.8	7.1	7.3	7.8	5.1	5.1	3.6	4.6	0.0	1.7	2.1	1.9
DE_2	9.3	8.4	7.6	7.6	6.1	7.3	7.4	6.9	4.8	6.2	4.1	5.7	1.4	0.0	2.4	3.1
DE_3	6.1	6.1	5.7	5.8	7.1	6.6	7.9	7.8	5.8	6.1	4.4	6.4	1.8	2.1	0.0	2.9
DE_4	5.8	6.4	8.1	7.4	8.0	7.9	8.3	8.6	6.4	6.7	6.3	7.1	2.7	2.9	2.6	0.0

US_1 = Buffalo
US_2 = Chicago
US_3 = Dallas
US_4 = Seattle

CA_1 = Montreal
CA_2 = Toronto
CA_3 = Calgary
CA_4 = Vancouver

GB_1 = Maidenhead
GB_2 = Milton Keynes
GB_3 = Edinburgh
GB_4 = London

DE_1 = Munich
DE_2 = Frankfurt
DE_3 = Berlin
DE_4 = Hamburg

A node might dispatch a part of the computation to two or more nodes depending on the computation. These nodes would then appear as siblings on the computation tree with the dispatching node as the parent. Therefore, the number of nodes and the depth of the tree is determined by the level of granularity achieved in the overall computation. A minimum spanning tree connects the UC lab servers with the country coordinators as discussed below.

We designate one node in each country as a coordinator node. This is not a permanent assignment and a number of factors as explained later governs the assignment. This coordinator node is determined by measuring the ping times from each of the servers in a particular country to our server at the university. An hourly measurement of ping times promotes the server with the minimum latency to the role of the country coordinator. This measurement ensures that the coordinator nodes receive the FFT tasks with minimum delay. The coordinator node then performs the role of a local leader and coordinates the messages exchanged between the peer nodes. Each coordinator node uses the distance matrix to determine its closest neighbors and distributes the tasks between them. Normally, we would expect the coordinator nodes to distribute the tasks to nodes within the same country since they should have the lowest intra node latency. However, experimental data bears the fact that sometimes nodes geographically distant regions also come together to perform a computation owing to variations in latency as discussed previously.

The coordinator node is also responsible for monitoring the messages exchanged in its neighborhood. At the start of a particular FFT, all messages pass through the coordinator node. However, due to the nature of the computations in Fast Fourier Transforms, as the computation proceeds, groups of nodes start to have more traffic between them. Once the local coordinator node detects that a set of nodes need to communicate more frequently, it can either decide to move a larger chunk of the work to one of the nodes or sets up a peer connection between these two nodes thereby bypassing the coordinator altogether and speeding up the computation. Nodes performing the computation periodically send heartbeat messages to the coordinator notifying their connectivity to the network. These messages also contain a marker that notifies the coordinator on the current progress of the task.

The coordinator can have an overview on the overall progress of the computation based on the progress of the individual nodes. This information sent to the servers at UC periodically allows us to monitor the performance of the system and tweak accordingly. In order to save the state of the tasks performed on the nodes, each node periodically saves the state of the computation as a checkpoint and updates the local coordinator. In case a particular node fails, the node coordinator can select another node to carry on the computation of the failed node from the last saved checkpoint. We can also mark certain sections of the computation as critical such that the nodes do a forced save of state before proceeding with the computation even if the normal scheduling does not call for a save of state. The coordinator simply copies the last checkpoint to the newly selected node and the computation

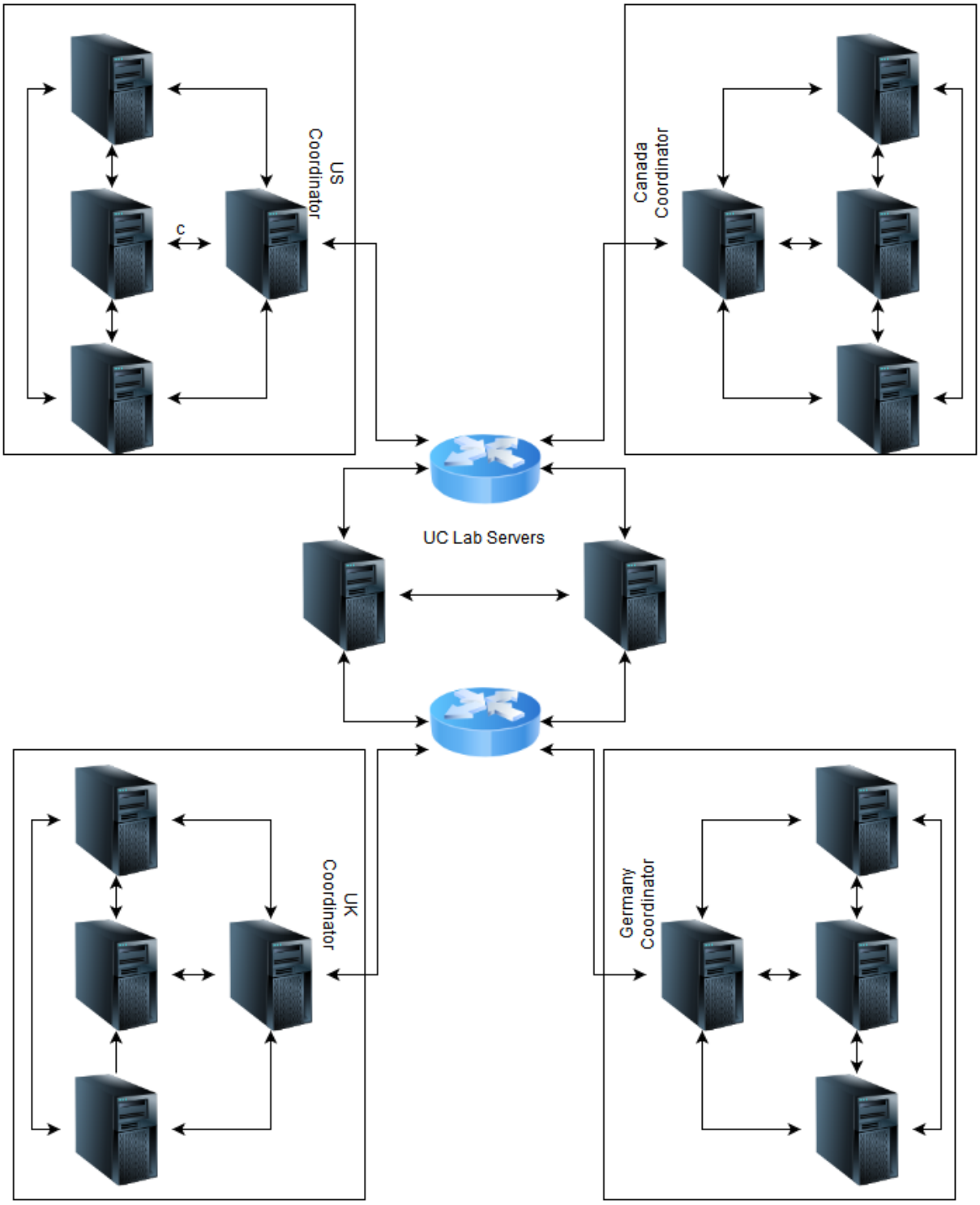

Figure 2. Schematic showing the connection between the nodes

can resume from that point onward. In order to ensure that the system can recover from the local coordinator failures, each local coordinator reports its status along with the status of all the coordinated nodes to the servers at UC. Additionally, we maintain a list of alternate servers that can take up the role of the coordinator in case of a failure. The process is similar to the mechanism that enables the coordinator to maintain track of all the associated nodes.

Additionally, the saved state of the coordinators determines coordinator node in case of increased latencies. Overall, the system always ensures that the nodes with the minimum latencies to UCs servers are the local coordinators. Having this hierarchical setup and periodic checkpoints ensures a high availability of the system with minimum of lost work in case of failures. Fig 2 shows the hierarchical setup as a schematic diagram.

3.3 Massively parallel Fast Fourier Transform

We would now show the computation of the Discrete Fourier Transform in parallel using the limited computing power of the devices in our system. The flow of control for the decimation in frequency for an 8-point Radix-2 FFT is in Fig. 3.

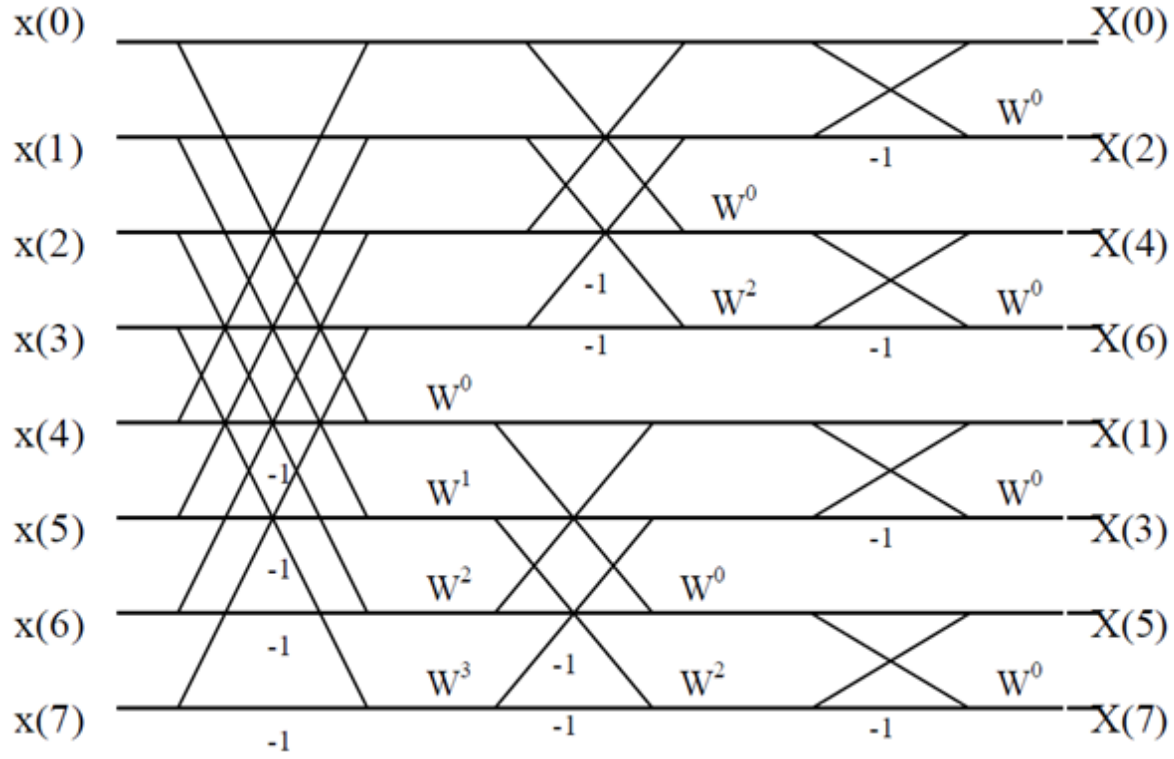

Figure 3. Decimation in frequency for the 8-point Discrete Fourier Transform [20]

From Fig. 3, we can see that the first stage involves a single 8-point Discrete Fourier Transform. As we move to the second stage, we have two distinct 4- point DFTs and eight distinct 2-point DFTs in the third stage. The corresponding decimation in time is in Fig. 4. The process for the decimation in time works in the reverse manner in that we start with a large number of 2-point DFT calculations and use the results of these computations as the input to the next stage. The important point to note is that while the complete DFT calculation would be computationally intensive, the computations required in each step of the decimation process are not. These typically require a few CPU cycles and computed by the low-powered CPUs in the IoT devices. The potential for massive parallelism is evident from Fig. 3 and Fig. 4 since only a group of nodes work on a particular stage of the decimation and therefore these steps execute independently.

We would now give a brief overview of the computational steps required for calculating

the FFT through decimation in time. Implementing the FFT computation through decimation in time involves the following steps:

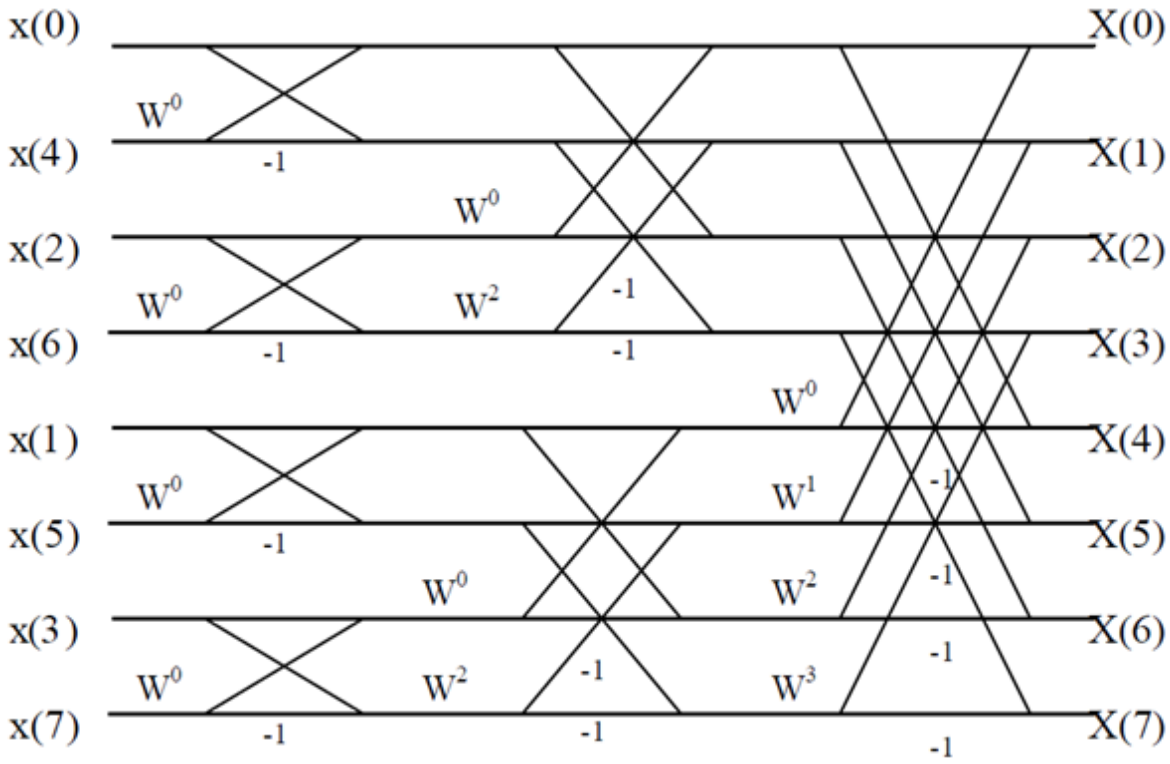

Figure 4. Decimation in time for the 8-point Discrete Fourier Transform [20]

1. Creation of the execution sequence map for the computation. From Fig. 4 we can see that the first stage involves 2-point DFT blocks. Hence, for an 8-point FFT computation we would have four, 2-point DFT blocks. The coordinator nodes at the start of the simulation broadcast this information. This ensures that the participating nodes are aware of the execution sequence and the nodes they should expect their upstream data. Preloading this information into the participating nodes also has the added advantage of failure tolerance. If one of the participating nodes goes down, the coordinator node knows exactly which part of the computation is lost and can therefore reload that part into another available node.
2. Each of the 2-point DFT blocks in the first stage has their outputs tied to one of the inputs of the 4-point DFTs of the second stage. We exploit our geographic localization to ensure these stages run in nodes that are close together.
3. Each of the 4-point DFT blocks now has their outputs tied to the input for the final 8-point DFT calculator.
4. Once the execution sequence map had been loaded, the simulation starts by invoking the first stage of the computation that involves the 2-point DFTs.

The process flow diagram for this decimation is in Fig. 5. Each of the ovals in Fig. 5 represents a unit of computation as shown in Fig. 4. The numbers inside the ovals represent the actual element numbers on which the computation runs. Labels under the oval designate the corresponding computing node. Since the process of Discrete Fourier Transform and its inverse are complimentary, we would have more parallelism in the first stage and the number of division halves in each successive step. However, we can only utilize one computing node in the first stage, since there is only one operation to perform. The processor utilization increases in the subsequent stages as the number of DFTs simultaneously increases. A severe disadvantage of this mechanism is the sheer number of message exchanges required in order to make the process run. From Fig. 5, we can see each point in the DFT calculation, message exchanges required in order to move to the next step.

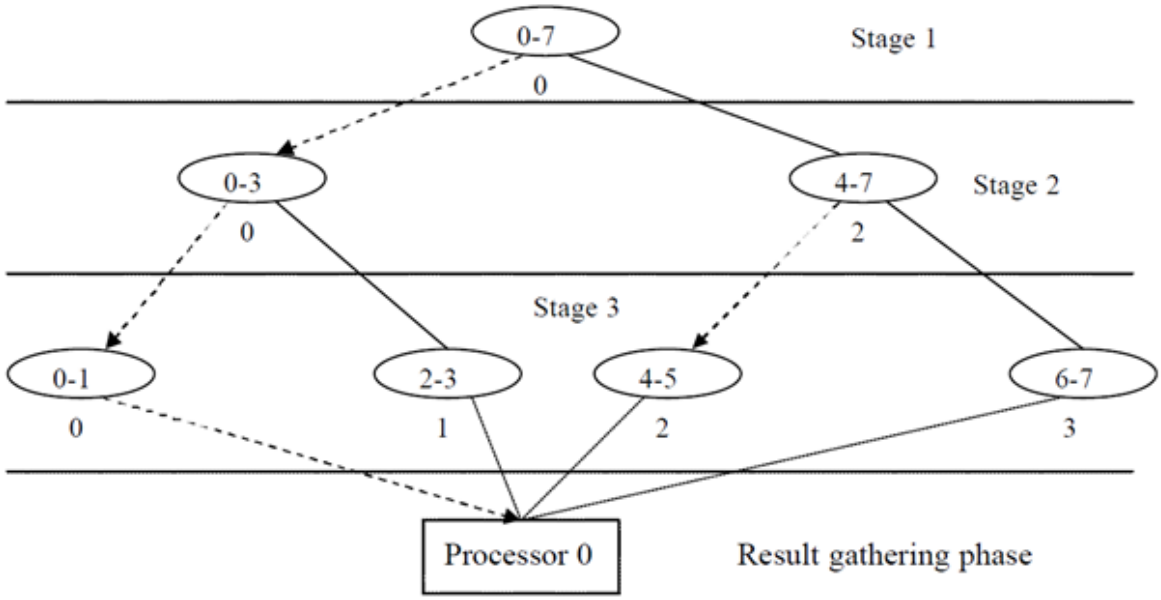

Figure 5. Process flow diagram during computation for the DFT [21]

A possible improvement is by reducing the number of message exchanges. We see that the node labelled “2” cannot proceed before the results from the first stage of the computation are available and it can proceed with the second stage. However, node 2 already knows about the data set that it is waiting on. Thus, it can perform the same computation that is running in node 0 and then use the results for the next stage of computation. In order to enable this out of order processing we need to perform a broadcast of the complete input data. Therefore, this method is very much suited for our experiments on the internet. While we understand that the approach poses a very real possibility that some nodes may end up performing redundant computations, it is a small price to pay since the real bottleneck in a distributed system on the internet is the network latency. Our own experiments have shown that the delay arising from this redundant computation is actually much lower in a real computation. A schematic diagram showing the communication lines for the 8-point Radix-2 FFT is in Fig. 6.

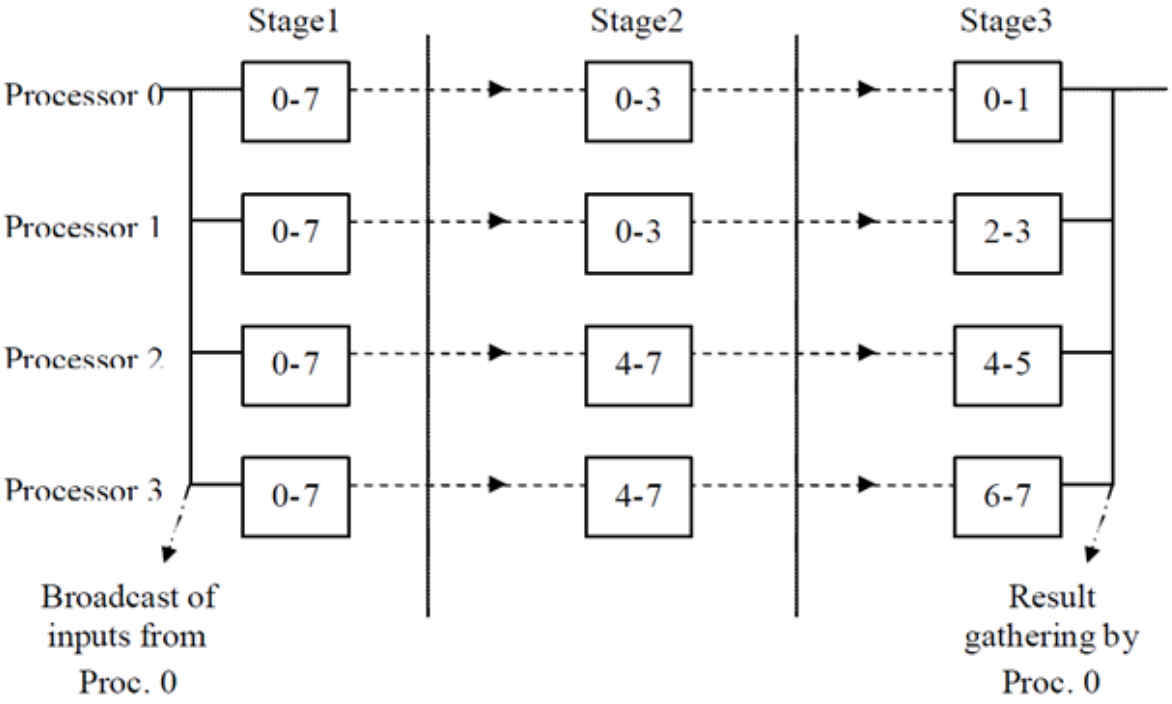

Figure 6. Control flow during the parallel computation of the DFT [21]

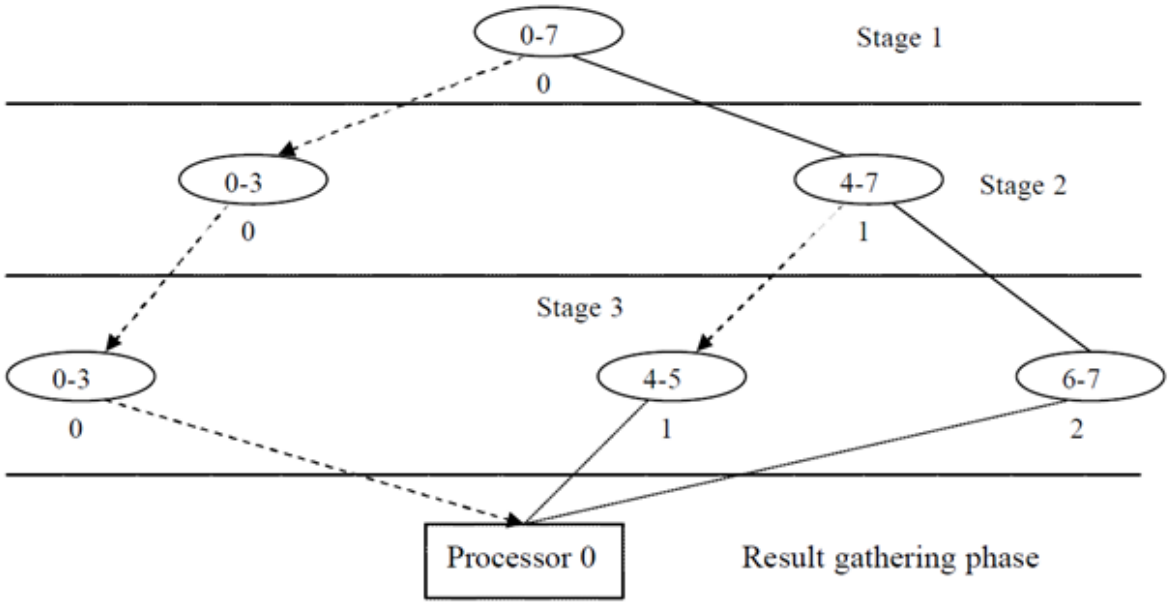

Figure 7. Process flow diagram for the DFT using three nodes [21]

It would seem that the mechanism works best when the number of computing nodes is a power of two. However, we have seen the mechanism implemented with any number of nodes. If, the processing elements were not exactly divisible by the number of nodes at any stage, then the node that was responsible for dividing the data set would continue to perform computations on the entire data set from the previous step. The process flow for three nodes is in Fig. 7. We have always assumed that computations would be performed utilizing the

spare capacity of the resources. Therefore, we have built our system from the ground up to work with a varying number of computing resources. As long as we have spare computing nodes available, and tasks are divisible, newly available nodes would be able to get a chunk of the work and contribute to the overall computation. Conversely, when nodes go offline, the local coordinator detects this condition from the message exchanges and moves the work set to another available node. The checkpoint mechanism enables speedy rollback and resumes. In order to obtain a steady throughput from our system, it should be stable enough to compute FFTs even when the number of computing nodes is not even. In general, the optimum FFT calculation requires a number of computing nodes based on the size of the input array. For most general-purpose FFT calculations, the optimum number of computing nodes for a given input FFT array size is in Table 2.

Table 2. Input array size and processor optimal parallel FFT [21]

Input Array Size	Number of Processors
1 – 256	1
256 – 16384	2
16384 – 1.5×10^5	4
1.5×10^5 – 1×10^6	8

4. Results

For the purposes of evaluating the performance, we implemented a multithreaded program in C++ to run these Fast Fourier Transform tasks on our globally distributed micro cluster. Each FFT computation spawns a new isolated process as long as the CPU and physical memory resources are available. We deliberately chose this isolated process approach in order to ensure that a crash in one running computation does not affect the other computations running on the same node. The results of the crashed process reported to the local coordinator reassigns it to another randomly chosen node in order to eliminate failures caused due to the setup of a particular node. If the chunk of work does not complete in three instances, it is permanently marked as failed and the overall computation is marked as failed. We perform an initial sanity check to determine the size of the optimal input. This allows us to get an estimate of the results expected from all the computing nodes at the end of the last stage. This computation is important since it gives our servers at the university a mechanism to determine the authenticity of the computed results. We determine granularity of the division of the task depending on the size of the input array and the number of nodes that are active at that point of time. The coordinator nodes then handle further network topology and traffic changes. The DFT computation is for both the upper and the lower halves or for either

of the halves since they are symmetrical. Computing one-half greatly improves the efficiency of the whole process. Our implementation gave us a chance to evaluate the impact that geolocation would have on the computational times and evaluate the speedups that can be achieved.

We ran multiple tests with Fast Fourier Transforms by varying the size of the input data array for FFT computations. Each test runs with a fixed array size for a million iterations in different times of the day in order to minimize transient network effects. As the network conditions were different for each run, we recorded maximum and minimum values of latencies and the overall achieved speedups. A combination of an optimum number of available nodes as well as the prevalent network traffic conditions can combine to give highly favorable results. Similarly, adverse network conditions and non-availability of computing nodes can give unsatisfactory results. Error bars in the graphs indicate the variance of our measured speedups. We have designated the average achieved speedup to be the observed speedup for each simulation. We have run a large number of simulations at different times of the day in order to counter the effects of network traffic. These have allowed us to have a higher degree of confidence on our results.

In order to determine the effect of geographic localization and our proximity matrix we split the tasks into two sets. In the first set of tests, we distributed the computation randomly across the cluster without any regard to the proximity of the nodes. In the second run, the node coordinators determine their closest neighbors in the same geographical location and selectively distributing the tasks to these nodes. Continuous evaluation of the proximity matrix and the associated overhead of exchanging messages to keep the matrix updated adds additional load to all the nodes performing the computation. For the purposes of our calculations we have added this small but finite overhead to the overall performance of the nodes. The speedups we have obtained take into account this extra overhead. We have tested the robustness of the checkpoint mechanism by introducing random failures in nodes and taking them offline. The results obtained are feedback for the system and used in order to improve the numbers in the proximity matrix. We varied the transmission rates of certain nodes in order to simulate heavy traffic scenarios. Additionally, we ran the tests during various times of the day in order to take advantage of the normal variations in internet traffic. Our results show significant improvements in performance through localization. We present our results in Figures 8 through 11.

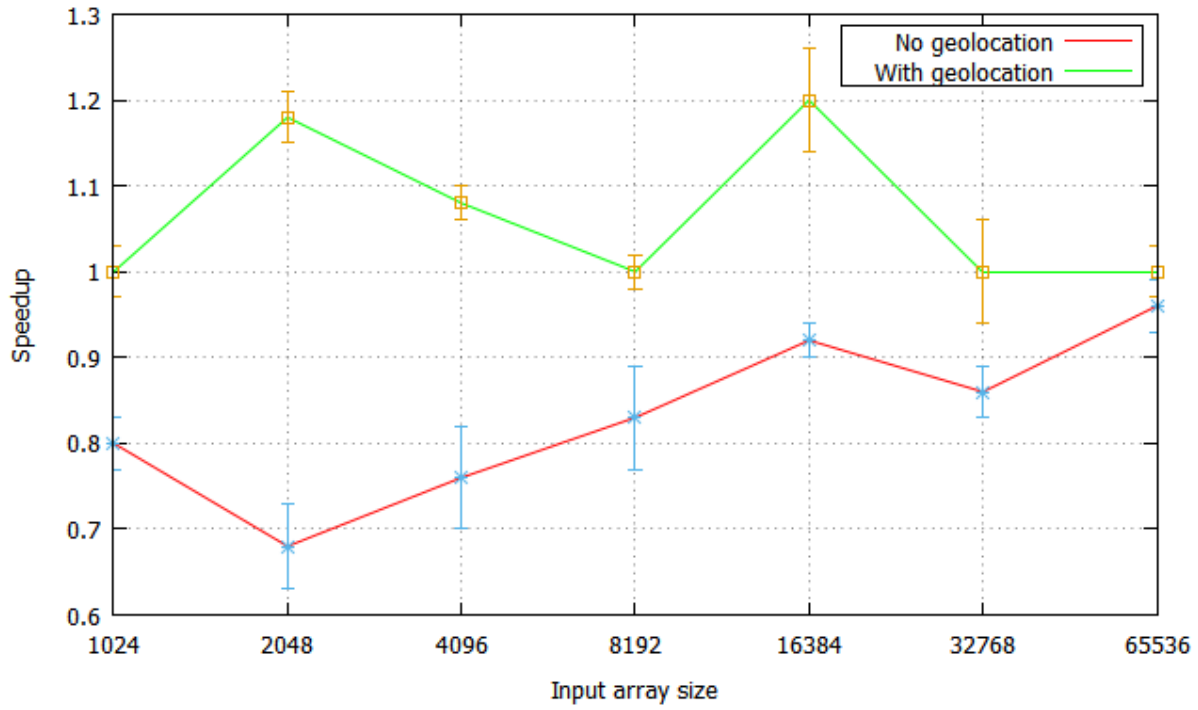

Figure 8. Achieved speedup with a single node

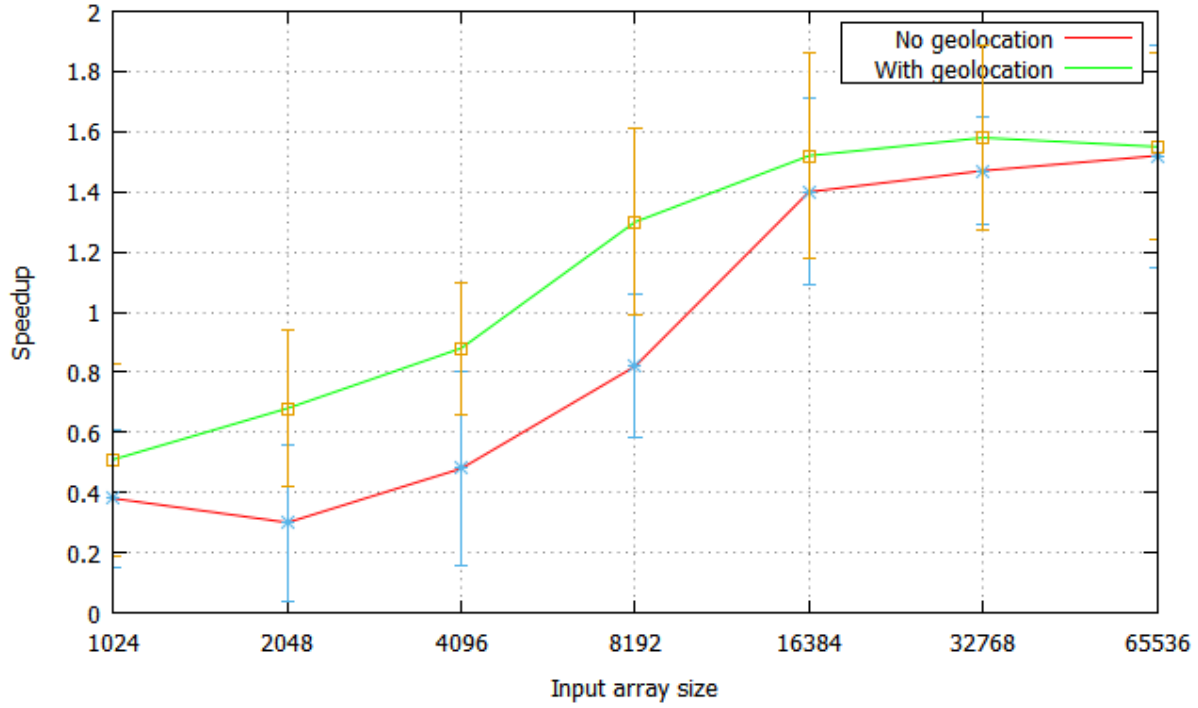

Figure 9. Figure 9: Achieved speedup with two nodes

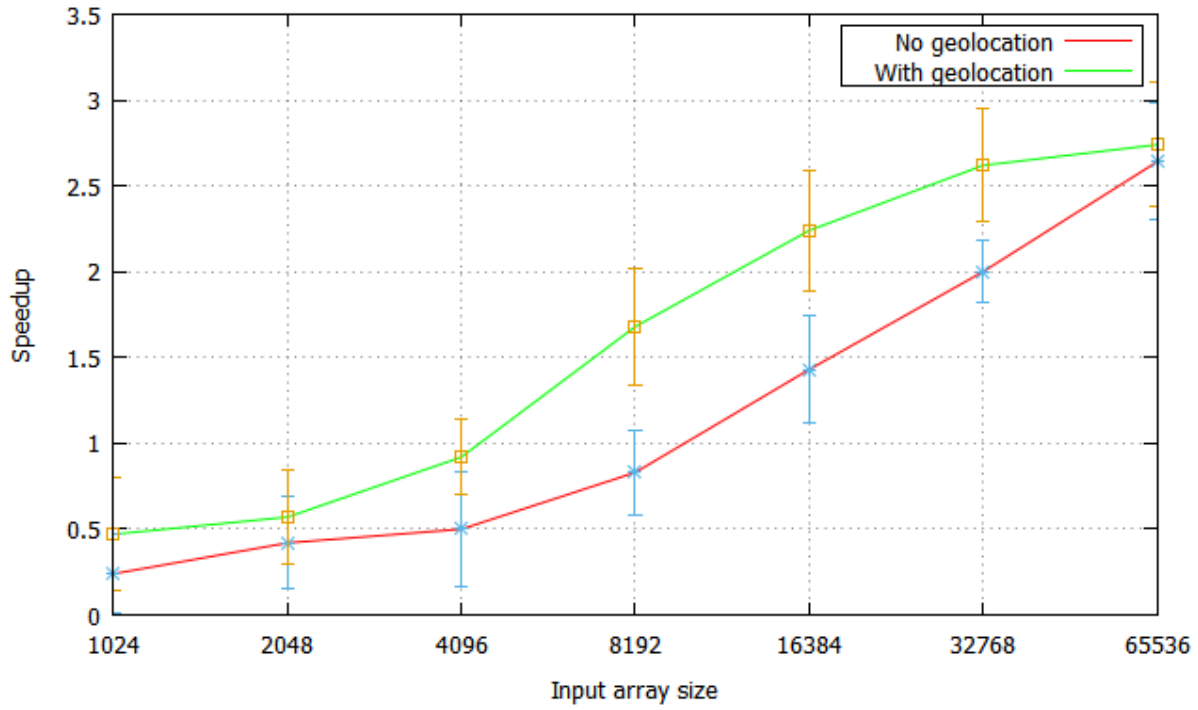

Figure 10. Achieved speedup with four nodes.

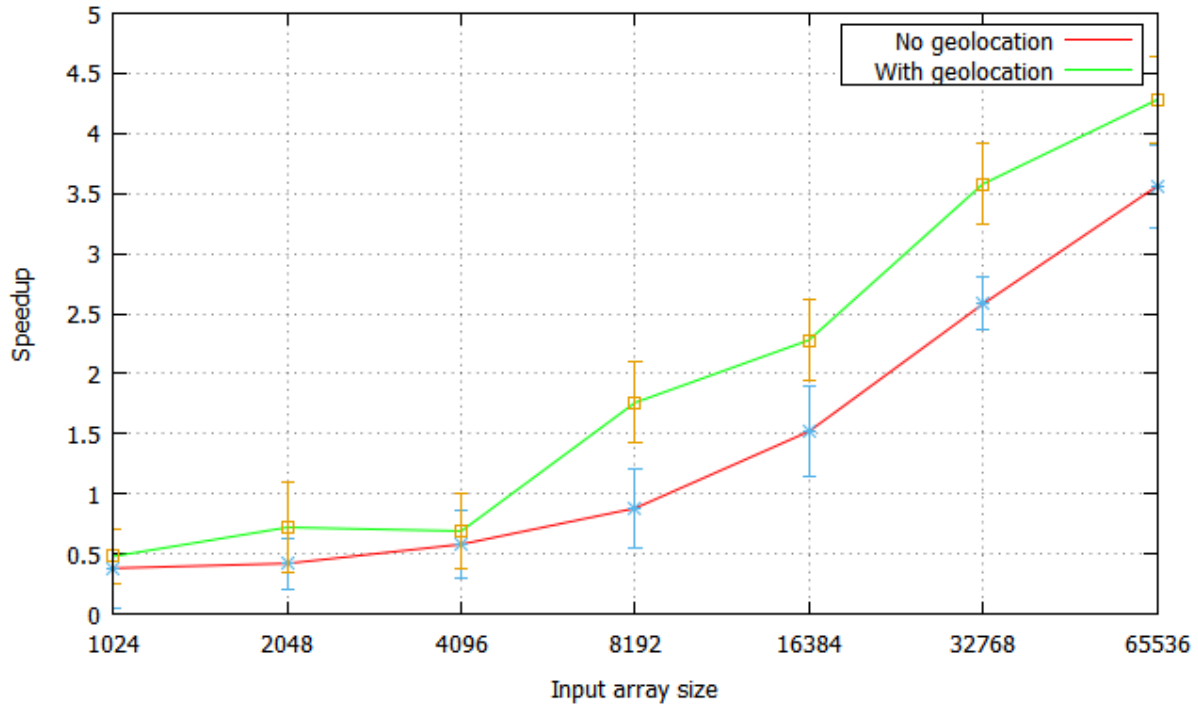

Figure 11. Achieved speedup with eight nodes.

5. Conclusions and Future Work

We observed that computing FFT offers a huge potential for massively distributed workflows. Combined with fine-grained parallelism, it enables us in efficiently distributing tasks to computing nodes with short turnaround times, which can also help us in recovering from failures easily. If we are computing an N-point FFT, we can have N parallel operations running provided we have enough computing nodes. This would be a significant boost in performance over traditional coarse-grained parallelism which requires that a large number of operations requiring substantial computing power at each node. The elastic nature of our system allows us to optimize the usage of available resources and ensures a high availability of the system.

From our observations, we can see that geolocation provides us with a better throughput since it actively selects nodes that are closest and distributes the task between them thereby reducing the latency in the communications required to get the computation completed. The single node test is a control test run with a single country coordinator to test the feasibility of our mechanism. In general, we have seen a few situations wherein the simulations have not progressed smoothly as in a dedicated cluster due to the non-availability of nodes and heavy network traffic. However, in almost all cases we have found that the system has recovered quickly by migrating the pending work to another available node. We attribute this to the small task divisions in fine-grained parallelism that makes it easier to distribute tasks and recover from failures quickly. We also observe that merely increasing the number of computing nodes does not always speed up the computation but directly relates to the size of the input array. Adjusting, the number of nodes taking part in the computation we can fine-tune the performance of the overall system thereby greatly improving its throughput.

We have currently run FFT as part of our simulation to test the feasibility and performance of our system. Our servers at the University of Cincinnati were fine-tuned to split the FFT tasks to the participating nodes. We plan to investigate the effects of fine-grained parallelism on dense linear algebra and encryption routines. The rationale behind it is to simulate the effects of non-regular computations on fine-grained parallelism. Our initial forays into tapping the latent power in low powered devices that comprise the internet of things have been successful. However, we have observed that most of the delays and blocks in the system occur due to nodes dropping out in the middle of a computation. We realize that this would be an ongoing problem in future IoT devices. Therefore, in order to make the system more robust and acceptable we are investigating on improving the rollback mechanisms and graceful failure recovery mechanisms. A reputation system that ranks nodes based on their turnaround times and successfully completed computations is in development. Once integrated with our system, nodes ranked higher would get more tasks allocated to them since there is a greater probability of the tasks completed in time. A common area of concern for all cloud systems is security. Studies have shown that almost all forms and types of cloud systems are vulnerable to at least some form of attacks [22]. We plan to expose our system through a set of APIs that would be accessible over the Internet. Interactive systems on the Internet are prone to a variety of exploits that have been well-researched [23]. We would be developing a comprehensive security framework to safeguard the computational capabilities

of our system from misuse. This would involve setting up cryptographic keys to authenticate all users of the systems as well as filters to remove malicious user payloads. All of these would enable our system to be more robust and greatly improve the overall throughput.

References

[1] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context aware computing for the internet of things: A survey," IEEE Communications Surveys Tutorials, vol. 16, no. 1, pp. 414–454, 2013. https://dx.doi.org/10.1109/SURV.2013.042313.00197

[2] A. H. Laine and S. Brahimi, "Background on context-aware computing systems," in Internet of Things and Advanced Application in Healthcare. IGI Global, pp. 1–31. https://dx.doi.org/10.4018/978-1-5225-1820-4.ch001

[3] E. Borgia, "The internet of things vision: Key features, applications and open issues," Computer Communications, vol. 54, pp. 1–31, 2014. https://dx.doi.org/10.1016/j.comcom.2014.09.008

[4] E. de Matos, L. A. Amaral, and F. Hessel, "Context-aware systems: Technologies and challenges in internet of everything environments," in Internet of Things. Springer Nature, 2017, pp. 1–25. https://dx.doi.org/10.1007/978-3-319-50758-3 1

[5] H. Sundmaeker, P. Guillemin, P. Friess, and S. Woelffl´e, Eds., Vision and Challenges for Realising the Internet of Things. Luxembourg: Publications Office of the European Union, 2010.

[6] S. C. Mukhopadhyay and N. K. Suryadevara, "Internet of things: Challenges and opportunities," in Internet of Things. Springer Nature, 2014, pp. 1–17. https://dx.doi.org/10.1007/978-3-319-04223-7 1

[7] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, ser. GRID '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–10. https://dx.doi.org/10.1109/GRID.2004.14

[8] E. Ivashko and N. Nikitina, "Web service of access to computing resources of BOINC based desktop grid," in Lecture Notes in Computer Science. Springer Nature, 2011, pp. 437–443. https://dx.doi.org/10.1007/978-3-642-23178-0 38

[9] C. B. Ries, "Berkeley open infrastructure for network computing," in Xpert.press. Springer Nature, 2012, pp. 27–33. https://dx.doi.org/10.1007/978-3-642-23383-8 3

[10] J. L. Hennessy and D. A. Patterson, Computer Architecture A Quantitative Approach, 4th ed. Morgan Kaufmann, 2007.

[11] S. Le Blond, A. Legout, and W. Dabbous, "Pushing bittorrent locality to the limit," Computer Networks, vol. 55, no. 3, pp. 541–557, Feb. 2011. https://dx.doi.org/10.1016/j.comnet.2010.09.014

[12] C. Decker, R. Eidenbenz, and R. Wattenhofer, "Exploring and improving bittorrent topologies," in Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on, Sept 2013, pp. 1–10. https://dx.doi.org/10.1109/p2p.2013.6688698

[13] T. Karagiannis, P. Rodriguez, and K. Papagiannaki, "Should internet service providers fear peer-assisted content distribution?" in Proceedings of the 5th ACM SIGCOMM

Conference on Internet Measurement, ser. IMC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 6–6. https://dx.doi.org/10.1145/1330107.1330115
[14] C. Dale, J. Liu, J. Peters, and B. Li, "Evolution and enhancement of BitTorrent network topologies," in 2008 16th Interntional Workshop on Quality of Service. Institute of Electrical and Electronics Engineers (IEEE), Jun 2008. https://dx.doi.org/10.1109/IWQOS.2008.6
[15] M. Su, H. Zhang, X. Du, B. Fang, and M. Guizani, "Understanding the topologies of BitTorrent networks: A measurement view," in 2012 IEEE Global Communications Conference (GLOBECOM). Institute of Electrical and Electronics Engineers (IEEE), dec 2012. https://dx.doi.org/10.1109/GLOCOM.2012.6503408
[16] The haversine formula to calculate distance between two points on the earth. http://www.movable-type.co.uk/scripts/latlong.html (Last accessed 3rd January 2017).
[17] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," Mathematics of Computation, vol. 19, pp. 297–301, 1965. https://dx.doi.org/10.2307/2003354
[18] I. J. Good, "Introduction to Cooley and Tukey (1965) an algorithm for the machine calculation of complex Fourier series," in Springer Series in Statistics. Springer Nature, 1997, pp. 201–216. https://dx.doi.org/10.1007/978-1-4612-0667-5_9
[19] The maxmind ip geolocation service. https://www.maxmind.com/en/home. (Last accessed 3rd January 2017).
[20] J. G. Proakis and D. G. Manolakis, Digital Signal Processing Principles, Algorithms and Applications, 3rd ed. Prentice-Hall International Inc., 1996.
[21] H. Karner and C. W. Ueberhuber, "Parallel fft algorithms with reduced communication overhead," Institute for Applied and Numerical Mathematics, Technical University of Vienna, Tech. Rep., 1998.
[22] J. S´anchez, G. Corral, R. M. de Pozuelo, and A. Zaballos, "Security issues and threats that may affect the hybrid cloud of FINESCE," Network Protocols and Algorithms, vol. 8, no. 1, p. 26, may 2016. https://dx.doi.org/10.5296/npa.v8i1.8727
[23] C. Yang, J. Ren, and J. Ma, "Provable ownership of files in deduplication cloud storage," Security and Communication Networks, vol. 8, no. 14, pp. 2457–2468, jul 2013. https://dx.doi.org/10.1002/sec.784

Copyright Disclaimer