Chapter 5 OptiFog: Optimization of Heterogeneous Fog Computing

IoT sensors assist in gathering real-time patient data which is time-critical concerning decision making. The frequency of sampling of the produced data and the density of sensors distributed across the area determine the volume of such data. This data is time-critical in nature i.e. the data will cease to have any worth if a timely decision is not taken. In case this data consists of essential health care information, it becomes additionally time-critical with respect to decision making e.g. continuous ECG tracking data. To process and store the data for future reference, this data is transmitted over the cloud, in the conventional IoT architecture. However, higher transmission delays cause redundant data to continually stream over the cloud and hence, hinder real-time decision making. The Fog Computing [105] solution is an alternative to this wherein data processing occurs in the LAN and only vital data and data usable for future reference, is transmitted over the cloud. The residual data is handled at the fog layer itself [106]. Fog Computing prevents redundant data transmission, saves network bandwidth and energy, and consequently reduces latency. Thus, if time-critical health care data is provided to the fog, it will result in quicker data processing which will contribute towards efficient medical facilities and subsequently, prevent patient mortality. On account of its computational limitations, fog layer cannot handle multiple data streams and thus, lags in terms of computing power. Distributed Computing in the Fog Computing framework can be used to conquer these shortcomings.

For a Fog Computing node, a Raspberry Pi (R-Pi) can be used [107]. A Raspberry Pi cluster can be utilized for improving the Fog Computing layer's overall computational abilities. Multiple Raspberry Pi with different configurations are assembled together to make a device called the Raspberry Pi cluster which acts as a Fog Computing node in this paper. For quicker computation, parallel processing is performed by the Raspberry Pi cluster which forms a distributed system. Raspberry Pi along with its cluster are discussed elaborately in subsection of section one.

“Dispy” is utilized on the Raspberry Pi cluster for implementing Distributed Computing. Parallel processing is enabled by ‘dispy’ across all nodes in the Pi cluster. n channels are created for n slaves and tasks are delegated to each slave via these channels, after which the reply is awaited. Subsequent to getting a reply on the channel, it delegates another task to the node and so on. In section two, dispy and Distributed Computing are talked about elaborately.

In the best-scenario (Big Omega (Ω) case), each algorithm gives the best possible outcome. An approach will certainly deliver on average (Big Theta (Θ)) and best-scenario if it produces optimum results in the worst- scenarios (Big Oh (O)). A comparative study/implementation of current techniques is carried out in this chapter and an approach that works effectively when other computing tasks are preloaded on all cluster nodes, is suggested. This is done to ensure that the proposed "OptiFog" algorithm will work effectively even in worst-case situations. In the next section, the working environment is discussed with respect to its heterogeneity, hardware and software configurations. A significant part of ECG signal analysis and its techniques are discussed elaborately in the same. The current chapter concerns computation tasks of analyzing an ECG signal and the same is performed by varying length ECG signals. The next section involves the installation of the dispy setup on all cluster nodes and the analysis of the ECG signals by the master-slave design. The sub job considers a collection of two ECG waves in the initial run and their computation readings are recorded. The sub job task is updated with respect to the number of ECG waves after analysis of these readings to minimise the overheads. Performance parameters for every one of these cases is described. Multiple logical and technical terms along with symbols and related functions are introduced and discussed.

Several parameters affect the overall computation of any computing system. Some such parameters like CPU usage, number of cores, memory and response time are recognized here. In order to comprehend the effect of such parameters in the given computation, the planned heterogeneous arrangement is operated against every one of these parameters and readings are documented. Every graph depicting every one of the parameter is thoroughly described post completion of the experiment. Section seven concerns these parameters and involves comprehending their subsequent impact on the computational task.

The suggested "OptiFog" algorithm is thoroughly described in the end section with its design and working. It involves explanation of the used parameters in the algorithm along with thorough discussion of performance improvement and expected performance outcome. Testing of the suggested algorithm is done on varied data sets in the last section for demonstrating its validity. Towards the end, the percentage of improvement and the Speedup factor are introduced.

5.1 Understanding the Working Environment

Varied hardware and software configurations are selected for performing the experiment. The requirement for various configurations along with their specifications will be comprehended here. Some computation is preloaded in every node of the system and the processing job along with details is described here.

5.1.1 The need for Heterogeneous Configuration

There are two setups which we use here and in both of them, the number of cluster nodes range from 1 to 4. Via techniques such as memory, response time, CPU usage and the number of cores, we shall comprehend the impact of computation-distribution. There are 2 types of nodes and the greatest number of nodes is 4 in a cluster. Considering hardware, one setup consists of all 4 homogeneous nodes whereas the other setup consists of 3 homogeneous nodes and 1 heterogeneous node respectively. This would assist in observing the overall performance effect, load distribution, effect of processing, and algorithm’s adaptability in a cluster, with diverse parameters and configurations.

5.1.2 Hardware Configurations of the cluster nodes

Raspberry Pi 3 model b+ [108] and Raspberry Pi 4 [109] models are utilized for the system with the following configurations:

Table 9: Hardware configurations of cluster nodes

Hardware Module	Raspberry Pi 3 Model b+	Raspberry Pi 4
Processor	Broadcom BCM2837B0, Quad-core Cortex-A53 64-bit SoC	Broadcom 2711, Quad-core Cortex-A72 64-bit SoC
Operating Freq.	1.4 GHz	1.5 GHz
Bluetooth	4.2	5.0
Wi-Fi	2.4 GHZ / 5.0 GHZ IEEE 802.11.b/g/n/ac/wireless LAN	2.4 GHZ / 5.0 GHZ IEEE 802.11.b/g/n/ac/wireless LAN
Memory	1GB LPDDR2 SDRAM	4GB LPDDR4 SDRAM
SD Card Support	Micro SD card	Micro SD card
Operating voltage & current	DC 5V/2.5A DC	DC 5V/3A

In comparison with the Raspberry Pi 3 b+ model, the Raspberry Pi 4 consists of better configuration. With respect to communication, memory, and processing, the Pi 4 node is better. For designing a suitable algorithm, the higher node is chosen deliberately in the system to observe the system behaviour across other parameters. Raspberry Pi 3 b+ is the dispy server node for the different nodes. The performance of the system depends on several computing parameters such as internal clock rate, bus speed, instruction set architecture, processor cache, and memory. Table 10 entails the details.

Table 10: Number of nodes and nodes selection

Experimental Configuration	Number of R-Pi 3	Number of R-Pi 4
1 node	1	-
2 nodes	2	-
3 nodes	3	-
4 nodes	4	-
4* nodes	3	1

Our arrangement, thus, comprises of 2 clusters: cluster 1 wherein all 4 nodes comprise of Raspberry Pi 3 b+ models and cluster 2 wherein 3 nodes comprise of Raspberry Pi 3 b+ model and 1 node comprises of Raspberry Pi 4 model.

5.1.3 Continuous Load on cluster nodes

To design the best suitable algorithm to process health care data efficiently in the Fog clustering environment, we have used worst case scenario where, each cluster node is kept occupied with some computational task other than processing of data concerning health care. The node is permitted to process data concerning health care by keeping the cluster node busy upto a specific CPU usage limit. For creation of system load in the Raspbian OS environment, the "stress" tool is utilized. "Stress" is a tool [110] utilized for testing performances of the system when loaded. It is utilized by the admin of the system for observing the performance of I/O syncs, cache thrashing, VM status, driver performance, CPU usage, and process termination and creation. According to the specification in the option field, this tool generates several types of load on the system. Until the said time elapses, it continues to generate that much stress on the system. In our case, the "sudo stress –cpu 1 –timeout 20000" command creates load of the CPU for required amount of time. It also keeps CPU occupied for about 25% for 20000 seconds, until all the algorithms are operated and tested in the system.

5.2 Processing ECG signal using a Dispy manner

For identifying the ECG intervals, the ECG wave sequence is separated into a set of 2 waves. The estimation whether the wave is abnormal or normal is made basis these intervals. In our case, every job allotted by the master dispy node to the slave nodes contains 2 ECG waves. The same procedure follows for different number of waves. Figure 5.1 below displays the time required for various number of waves by various nodes.

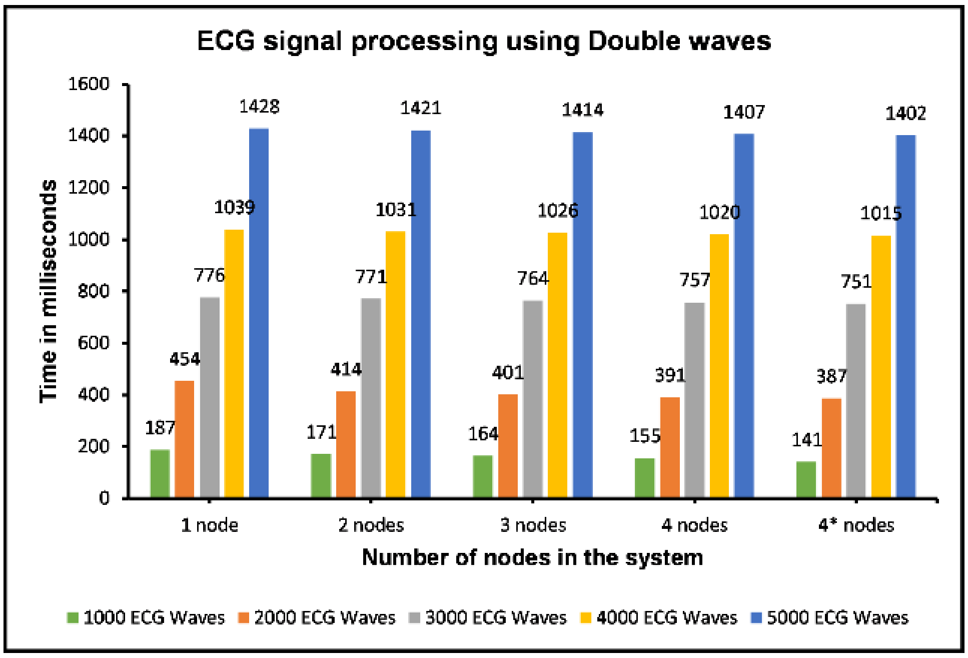

Figure 5.1 ECG signal processing using two ECG waves in one sub job

The dispy master node needs to perform multiple iterations for task completion in the system set up providing results in figure 5.1. Each iteration involves socket communication, communication overhead, and task distribution work. 1392 bytes is the mean size of the ECG wave. Bits per second are used to quantify the network performance for any communication system [111]. 84 bytes and 1538 bytes are the minimum and maximum frame sizes respectively, for Layer-2. 46 bytes and 1500 bytes are the permissible payload that can be carried by each frame. So if we allow maximum number of waves then it will allow larger frames and hence by reducing the number of frames. Consequently, there is lesser overhead of 38 bytes header per frame. Hence, multiple number of waves are used in a singular job to minimise the number of iterations.

The details concerning processing time for multiple nodes when every sub-job of the Slave node carries 4 ECG waves are shown in figure 5.2.

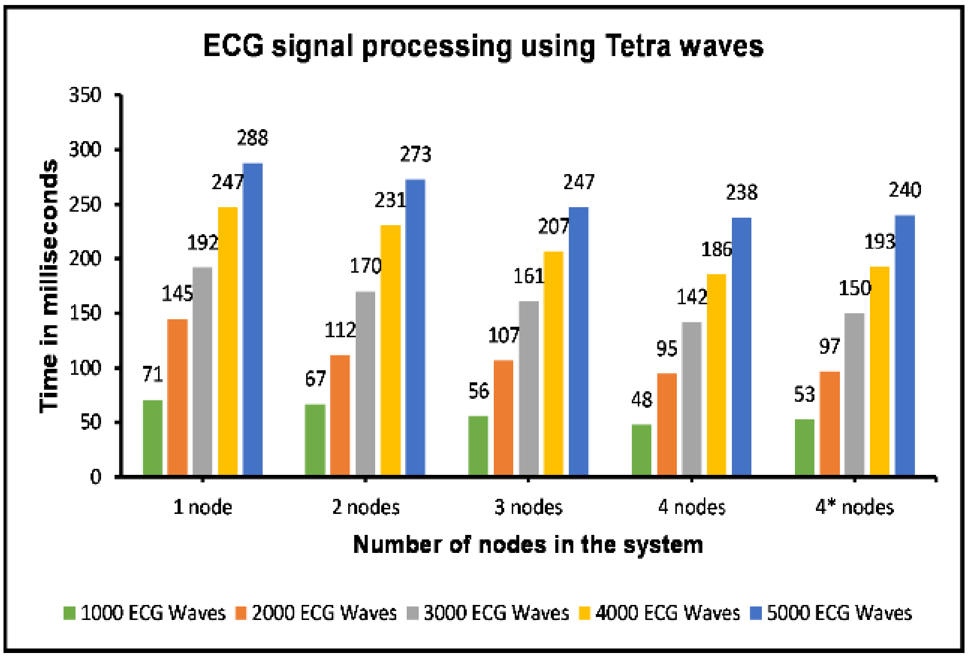

Figure 5.2 ECG signal processing using four ECG waves in one sub job

It is found that the dispy system computes the task in relatively lesser amount of time when 4 waves are utilized in sub-jobs. This is on account of lesser iterations that generate lesser delays and overheads. 8 ECG waves are used to probe further and minimise processing time. Now, each sub-job sent by master-dispy to the slaves' dispy constitutes 8 sets of sequential ECG waves from the main job. Figure 5.3 displays the processing time details for all waves. We observe that the performance of the system enhances considerably when the set of 8 waves is utilized. It considerably minimises the time taken for processing which is on account of the lesser overheads created in the dispy system.

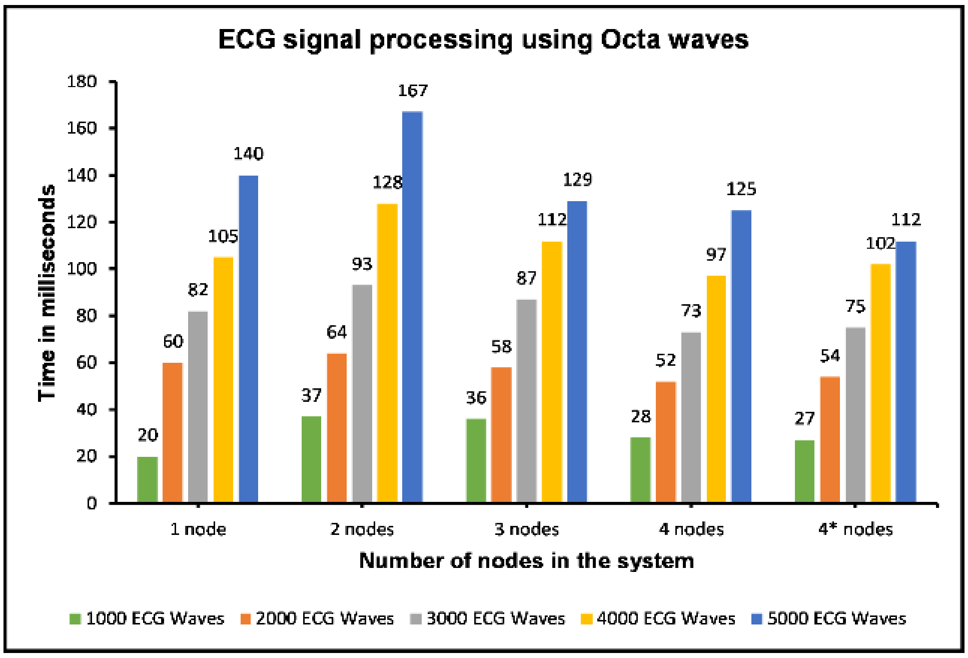

Figure 5.3 ECG signal processing using eight ECG waves in one sub job

We, further, test the set of ten ECG waves in a single sub-job for further minimization of processing time. Figure 5.4 entails the measure of performance for the same.

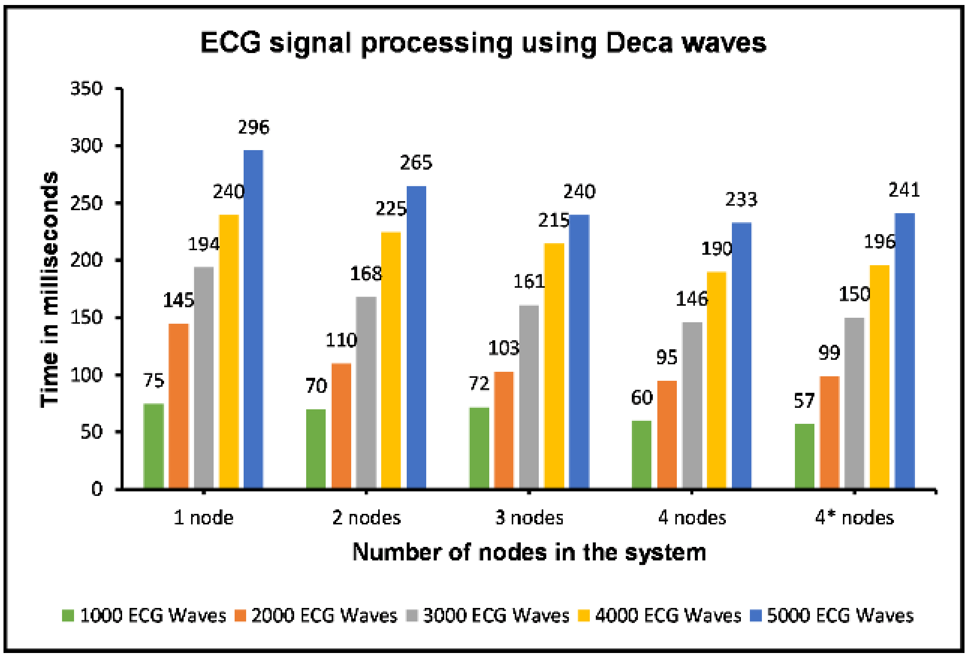

Figure 5.4 ECG signal processing using ten ECG waves in one sub job

We observe that there is no difference in performance in the 10 wave case and the 8 wave case. This is because the cluster nodes are preloaded with computation and accept 10 waves for processing in a single job. More computations take place on the pre-loaded nodes due to the higher amount of data carried by such 10 waves and this consequently increases the computational time for every sub job. Thus, the communication overhead and take distribution are lower in the 8 wave case than in the 10 wave case but the computational time is higher in the latter which is why the 10 wave case does not function as anticipated. Thus, we take the 8 wave sub-job instance as reference and performed more enhancements on it.

5.3 System Symbols, Functions and Performance indicators and their implications

Detailed beneath are the different symbols, terms and functions in the suggested system.

5.3.1 System Functions

Table 11: System functions and description

Functions	Description
timestamp sendTask(sub job, slave_id);	sends sub-job data to a particular slave bearing the mentioned id and returns the timestamp of that event
timestamp receiveTask(result, slave_id);	receive the result of the given task from the slave having the id number and note the timestamp of that event
timestamp getTimestamp( );	returns timestamp
CPUUsageQueryCU(slave_id);	returns CPU usage of slave_id
MUsageQueryMU(slave_id);	returns memory usage of slave_id
NC QueryNC(slave_id);	returns average availability of cores of slave_id
timestamp getFactors(subjob, slave_id);	returns timestamp when giving sub job to slave_id
string_objectreceiveFactors(job_id, slave_id);	returns string_object after processing job_id on slave_id
calculate(Ͼ, Ł, µ);	calculates capacity, time and memory factor

5.3.2 System parameters

Table 12: System symbols and description

Symbol	Description
J	Complete available job at the master node
J_i	Sub job of J, $J_i \subseteq \{J\}$: sub job with i ECG waves
M,S	Master Node, Slave Node
S_i	ith Slave Node
n	Number of available slave nodes in the dispy system
X_i	Xth factor for ith slave node
S,r	Sub job in sending context, Sub job in receiving context
T_s	Sending time of a particular sub-job on the master node
T_r	Receiving time of a particular sub-job on the master node
T_{is}	Sub job sending time to ith node from master node
T_{ir}	Sub job receiving time of ith node on the master node
T_{ijk}	T is timestamp value, where i: is slave node number, j: is the sub-job number, k ϵ {s,r}
N	Total number of sub-jobs
J_{ci}	Result of completed sub-job by ith node
$SelectNode_i$	ith node is selected for the further set of sub-job processing
RT	Response time
Ƞ	Ƞ ϵ {normal, abnormal}, i.e., the characteristic of ECG wave
CU, MU	CPU usage, Memory usage
NC	Average of available core % on the node
∀	For all
RT_o	sum of all RT_i , i∀ {1,2,…n}
MU_o	sum of all MU_i , i∀ {1,2,…n}
O_i	string_object having values timestamp, MU_i, CU_i, and NC_i
O_{ijr}	i: is slave number, j: is the sub-job number and r: is receiving context
O_r	Returning object by ith slave
Ᵽ	Priority factor
Ꞓ	Capacity factor
Ł	Time factor
µ	Memory factor
ψ	Impact factor
α	Number of jobs in one go, where each job has set of eight ECG waves
α_c	Result of completed α jobs
J_α	Sub jobs with α number of jobs
*	Multiplication
ms	Milliseconds
P_{ACO}	Performance time taken by Response Time technique
P_{CU}	Performance time taken by CPU usage technique
P_{MU}	Performance time taken by memory usage technique
P_{NU}	Performance time taken by a number of cores technique
$P_{OptiFog}$	Performance time taken by OptiFog algorithm
slave_id	Number representing a slave
job_id	Number representing a sub job

5.4 Finding different optimization parameters and using them in the system to understand its effect on computation

5.4.1 Applying Response time in the proposed System

Response time is comprehended and its importance is understood prior to using it as a system criterion. [112-113].

There are several time measures in an OS:

1. Arrival time (AT) – The time at which the process enters the ready queue.

2. Waiting time – The time spent by the process in the ready queue, waiting for execution.

3. Response time (RT) – The time between the arrival of the process inside the ready queue and its execution/entrance into the processor for the first time.

$$RT = \text{time when process gets first CPU} - AT \qquad (1)$$

The time is taken between the submission of a request and the very first response to that particular request. For good CPU scheduling, algorithm response time should be minimum.

4. Burst time – Time for which the process is under execution in CPU.

5. Completion time – Time at which the process is completely executed.

6. Turn Around Time – The total time required by the process to be executed and it is including the waiting time.

Response Time is crucial for interactivity. However, there may be a concern regarding more importance of how quickly the whole process is carried out rather as compared to the first time the process is served. As the job requires quick start and no interruption in the middle for completing the process at the earliest, this idea is inherently irreconcilable. The aim of the suggested system is the minimisation of the job computation time via usage of Raspberry Pi cluster and the Distributed Computing approach with the use of Dispy. The master node assigns the target sub-jobs to all the nodes and the finished jobs are received from the slave nodes. The master node tracks and maintains all the timestamps while assigning and receiving the jobs, the difference of which is used to calculate the response time of the node. Thus, the better node is selected for the job on the basis of these response times (RT), and the node with the best response time is allotted the residual jobs. Following is the algorithm.

Algorithm 2 Considering the Response Time to process the Job J

```
    Input: Complete job J having continuous ECG waves
    Output: Ƞ for each and every wave
1.  procedure find_ACO_Node
2.  $T_{is} \leftarrow$ getTimeStamp();
3.    for i ← 1, 2, ....... n  do
4.      $T_{ijs} \leftarrow$ sendTask( $J_i$ , $S_i$ );
5.      $T_{ijr} \leftarrow$ receiveTask($Jc_i$ , $S_i$);
6.      $RT_i \leftarrow T_{ijs} - T_{ijr}$
7.    end for
8.  $selectNode_i \leftarrow \min(RT_1, RT_2, \ldots RT_n)$;
10.   for j ← n+1, n+2, ....... N do
11.       $T_{ijs} \leftarrow$ sendTask( $J_i$ , $selectNode_i$ );
12.       $T_{ijr} \leftarrow$ receiveTask($Jc_i$ ,
    $selectNode_i$);
13.   end for
14. $T_{ir} \leftarrow$ getTimeStamp();
15. $P_{ACO} \leftarrow T_{ir} - T_{is}$
16. end procedure
```

The P_{ACO} is calculated for different number of nodes each time and the same is displayed in the graph below.

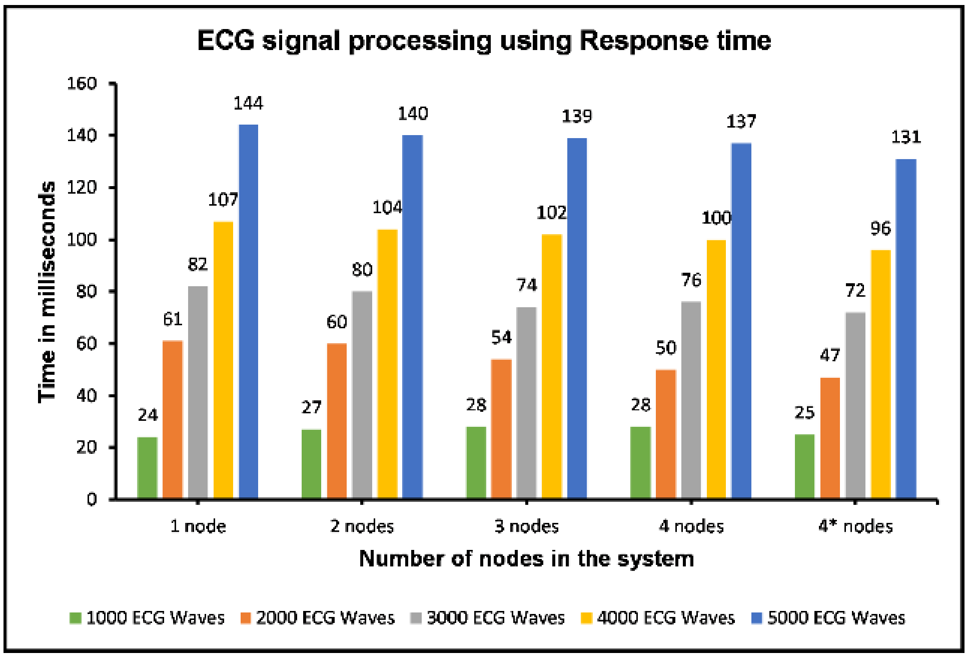

Figure 5.5 ECG signal processing using response time criteria

Graph Interpretation:

- After a specific time, the number of nodes is irrelevant as we observe that the performance in 3 nodes and 4 nodes is almost alike.
- The effect of 4* node is seen in the performance.
- Due to the overhead of finding the best node, the performance in 1 Node and 2 Nodes is not that enhanced.
- Due to the overhead of finding the smallest Response Time, the performance for 4* Nodes is below the Graph 4 observations.

5.4.2 ECG signal processing by CPU Usage, Available memory and Number of free cores

Genetic Algorithm states "Survival of Fittest". The fitness in the suggested system is identified with the help of various parameters such as:

- CPU Usage
- Available Runtime memory
- Number of available free cores in the node

The above mentioned parameters are extremely crucial as they have a significant effect on any processing task. For comprehending the effect of the parameters on the suggested system, every parameter is considered and experimentation is carried out for different number of nodes and waves.

5.4.3 ECG signal Processing based on CPU Usage

The logical, arithmetic and input/output control operations are executed by the Central Processing Unit (CPU). The CPU's cardinal operations at the time of executing a process are fetch, decode and execute [115], which together constitute the instruction cycle. The information is fetched from the memory after the program counter determines the fetch address. The Program counter hops to the subsequent instruction present in the sequence, post fetching. The fetch address is kept in the Instruction register. Prior fetched CPU instructions are interrupted to determine the subsequent operation of the CPU on the basis of the instruction, in Decode. As there are distinctive opcodes for each instruction, the same are decoded to determine which operation needs to be executed subsequently. The architecture of the CPU determines to execute a single action or a sequence of actions, in Execution. To enable fast access for the very subsequent instructions, the intermediate results are kept in the register of the CPU. On the basis of the process detailed above, the CPU time is calculated as

$$\text{CPU time} = \text{Instruction Count} \times \text{Clock Cycles/instruction} \times \text{Clock Cycle time} \ldots\ldots\ldots\ldots(2)$$

Additionally, on the basis of different CPU architectures, these fundamental instructions i.e. fetch, decode and execute can be divided into sub-operations for attaining a better level of pipelining. Numerous buses are used by different computer devices such as Memory, CPU, I/O and others for communication with one another. The main buses used in execution of process are as follows:

Address bus: Address bus carries the address of the data to be accessed or written from the processor to the memory.

Data bus: Data bus carries the data from the processor to the memory and vice versa.

Control bus: The basic instructions are read/write the data, and these control signals are given by the processor to the memory via the control bus.

As the CPU usage is computed each time, the node with highest CPU availability is allotted the job i.e., the node where the CPU usage is lowest because CPU usage plainly indicates the level of busyness of the CPU and lowest usage will allot the task to the relatively free node.

Algorithm 3 Considering CPU usage to process the Job J

```
        Input: Complete job J having continuous ECG waves
        Output: ɳ for each and every wave
1.      procedure find_CPU_Usage_Node
2.       $T_{is} \leftarrow$ getTimeStamp();
3.         for i $\leftarrow$ 1, 2, ....... j do
4.           for i $\leftarrow$ 1, 2, ....... n do
5.              $CU_i \leftarrow$ queryCU($S_i$);
6.           end for
7.          selectNode$_i \leftarrow$ min($CU_1$, $CU_2$, ..... $CU_n$);
8.          $T_{ijs} \leftarrow$ sendTask( $J_i$ , selectNode$_i$ );
9.          $T_{ijr} \leftarrow$ receiveTask($Jc_i$ , selectNode$_i$);
10.       end for
11.     $T_{ir} \leftarrow$ getTimeStamp();
12.     $P_{CU} \leftarrow T_{ir} - T_{is}$
13.     end procedure
```

Figure 5.6 displays the performance time on the basis of CPU usage.

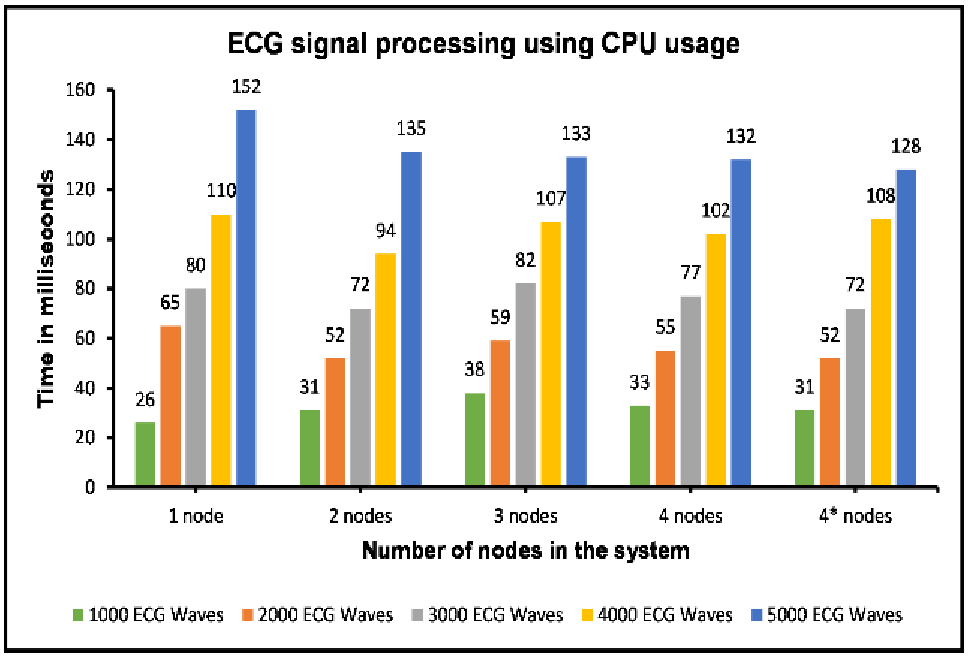

Figure 5.6 ECG signal processing using CPU usage criteria

Graph interpretation:

- The performance is almost alike for 1 node and the dispy system.
- Increase in number of nodes are not helping to improve the performance as it is causing query() to overhead on the system.
- 4* nodes system shows improvement.

5.4.4 ECG signal Processing based on available main memory

Widely, 2 objectives need to be attained as the OS manages the computer's memory [116]:
1. The memory space for each process must not lap over another process' memory space and there must be sufficient memory space for the execution of each process.
2. For the effective execution of each process, memory in the system must be appropriately utilized.

Process refers to a program under execution. A program is located on the disk and is run in the main memory and hence, must be relocated to the main memory from the disk. From a computation perspective, a process is described on the basis of its CPU state, execution environment and memory contents. Several registers like Stack Pointer (SP), Program Counter (PC), Instruction register (IR), and general-purpose registers are used to define a CPU state. The code of the program and predefined data structures reside on the memory. Reserved memory region which is designated to the program for run-time dynamic memory allocation is referred to as Heap. Return values of function calls, local variables and even few register values are stored in the Stack.

What does a process look like in memory?

The process memory is split into four sections [113]:

- The text section consists of the compiled program code, which reads in from the disk storage when the program is executed.
- Data section consists of the static and global variables which are stored before executing the main function.
- The heap is used for dynamic memory allocation. This data structure is managed by using calls to new, free, malloc, delete, etc.

The stack is used for local variables. The stack is reserved for local variables as and when they are declared. When the variables go out of scope, this space becomes free. It is also used to store the return values of the function.

Beginning at the contrary end of the free space of the process, the heap and stack proceed towards one another. In an ideal scenario, they should not meet. However, if they do, then either a stack overflow error shall take place due to inadequate memory or a call to new or malloc shall fail. Various address spaces are occupied by various processes in the main memory. Memory management is performed by the memory manager. In order to make space for the other processes in the main memory, program post completion need to be transferred out of the main memory. Amount of main memory used by the system is also referred to as memory usage. The average memory-access time for the process is computed as

Average memory_access time = Hit rate + Miss rate × Miss Penalty………..(3)

The algorithm to process ECG signal using available main memory is as follows.

Algorithm 4 Considering Main Memory to process the Job J

```
     Input: Complete job J having continuous ECG waves
     Output: Ƞ for each and every wave
1.   procedure find_min_MainMemoryUsage_Node
2.    $T_{is}$ ← getTimeStamp();
3.   for i ← 1, 2, ……. j do
4.     for i ← 1, 2, ……. n do
5.       $MU_i$ ← queryMU($S_i$);
6.     end for
7.   selectNode$_i$ ← min($MU_1$, $MU_2$, ….. $MU_n$);
8.   $T_{ijs}$ ← sendTask( $J_i$ , selectNode$_i$ );
9.   $T_{ijr}$ ← receiveTask($Jc_i$ , selectNode$_i$);
10.   end for
11.  $T_{ir}$ ← getTimeStamp();
12.  $P_{MU}$ ← $T_{ir}$ - $T_{is}$
13.  end procedure
```

The performance effect of the memory usage criteria is as displayed beneath.

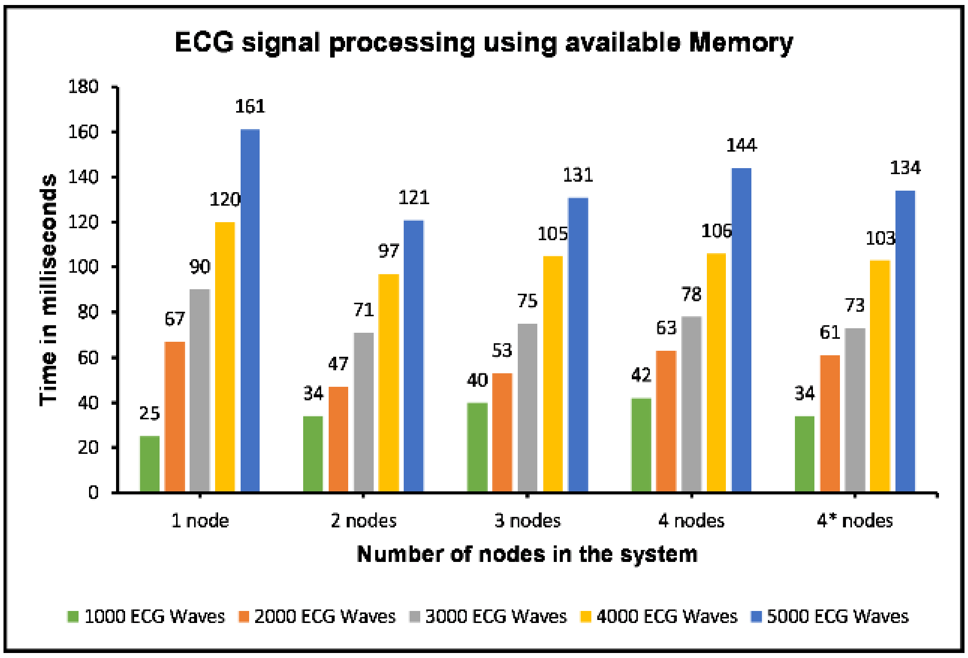

Figure 5.7 ECG signal processing using available main Memory criteria

Graph Interpretation:

- Performance does not enhance with increasing nodes.
- Less effective than other used techniques.
- Querying for available memory is the more overhead for more nodes
- Impact of 4* nodes can be observed.

5.4.5 ECG signal Processing based number of cores

A CPU may consist of more than a single processing unit, wherein each such unit comprises independent Arithmetic and Logical Unit (ALU), registers and control unit [117]. These processing units are called "core". Today, CPUs come with 8, 10, and even greater number of cores. Since all the cores can function concurrently, they perform the processing job quicker and thus, assist the CPU. For enabling communication of data, there exist communication channels betwixt the cores. These channels are arranged in the form of a mesh topology i.e. 6 communication channels between 4 cores. The performance of the CPU improves with higher number of cores [118-119]. To identify the number of cores and their specific percentage usage, we query each node. Further, to determine the availability of the core, the usage is then subtracted from 100. Similarly, availability of all the cores is added and the sum in then divided by the number of cores in the node which gives the NC value. The node with the highest NC value is selected as the node with highest number of cores available.

Algorithm 5 Considering the free number of cores to process the Job J

```
        Input: Complete job J having continuous ECG waves
        Output: Ƞ for each and every wave
1.      procedure find_number_of_Free_cores_Node
2.       $T_{is}$ ← getTimeStamp();
3.      for i← 1, 2, ……. j do
4.         for i←1, 2, ……. n do
5.           $NC_i$ ← queryNC($S_i$);
6.         end for
7.          $selectNode_i$ ← max($NC_1$, $NC_2$, ….. $NC_n$);
8.          $T_{ijs}$ ← sendTask( $J_i$ , $selectNode_i$ );
9.          $T_{ijr}$ ← receiveTask($Jc_i$ , $selectNode_i$);
10.       end for
11.     $T_{ir}$ ← getTimeStamp();
12.     $P_{NC}$ ← $T_{ir}$ - $T_{is}$
13.     end procedure
```

Figure 5.8 displays the performance measure of the above mentioned criteria.

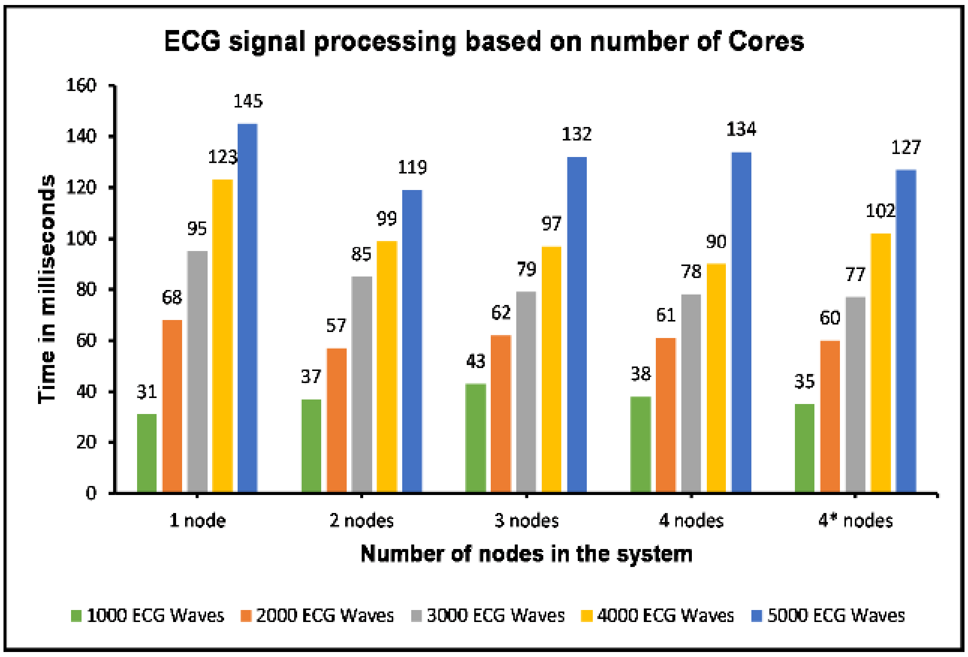

Figure 5.8 ECG signal processing using a number of Cores criteria

Graph Interpretation:

- We observe processing enhancement as the number of nodes exceed one.
- We observe query overhead.
- Performance enhancement is observed with the present of 4* node.

5.5 OptiFog algorithm

In the heterogeneous case, the suggested idea is expected to work ideally by leveraging the highest available processing power in the system. Response-time-based, memory-based, number-of-cores-based, and CPU-usage-based are the 4 techniques utilized. While each node was occupied with some other computational work, each technique was tried. This is done for figuring which technique is more weighted and less weighted in computing scenarios which are heterogeneous. On analysis of the obtained results and graphs for greater jobs and higher nodes, it is discovered that the best results are provided by the CPU-usage-based technique followed by number-of-cores-based technique. Response-time-based technique stands third and memory-based technique last.

5.5.1 Insights to the OptiFog Algorithm

OptiFog is an algorithm for hybrid optimization which is used to determine the impact factor on the basis of the 4 above mentioned techniques. The impact factor determined here is the value of each node. The number of jobs assigned to every node in a single go on the basis of this value. The existing status of Cores, CPU usage, and Memory is sent to the master node by every node. The number of allotted jobs is on the basis of the impact factor (ψ) which is calculated by the Master node in each iteration along with the Response time. On the basis of the processor specifications, operating frequency, bus size, and cache size, the cores and CPU on each node have varying capacities which depict a node's processing capabilities. This makes for another important reason for assigning higher priority (₽) to this factor. Whereas the memory and response time of a particular node can be compared with other nodes in terms of size and unit like GB and ms. Therefore, the response time and memory are observed as collective units in the distributed system.

OptiFog utilizes 3 important factors - Capacity (Ꞓ), Memory (µ) and Time (Ł), for determining the impact factor which is nothing but the node's total capability considering CPU, memory, response time and cores.

1. Capacity Factor (Ꞓ): It is based on the CPU usage and number of idle cores technique. The usage of the CPU is relevant to the number of cores. Both these techniques provide almost similarly fine performance. Thus, the factor is computed by using these 2 values as

Ꞓ = NC* (1-CU),where CU ϵ [0,1]…………………(4)

2. Memory Factor (µ): The node functions better if MU_i is less for this factor. Thus, for each node, MU_i is determined and further scaled to 1. This factor is calculated as

$$\mu = \frac{(1-MU_i)}{\sum_{i=1}^{n} MU_i} \quad \ldots\ldots\ldots\ldots\ldots\ldots(5)$$

3. Time Factor (Ł): Bearing in mind that a node's response time is inversely proportional to its capacity, the factor is formulated in a way that the node with lesser response time will be assigned a higher rank and the node with higher response time will be assigned higher ranks.

$$Ł = \frac{\sum_{i=1}^{n} RT_i}{RT_i} \quad \ldots\ldots\ldots\ldots\ldots\ldots(6)$$

The final ψis computed after determining Ꞓ, µ and Ł as

$$\psi = 3Ꞓ + 2Ł + 1\mu \quad \ldots\ldots\ldots\ldots\ldots\ldots(7)$$

where, numericals ϵ {Ᵽ}

Following is the OptiFog Algorithm:

Algorithm 6 OptiFog Algorithm to process the Job J

```
    Input: Complete job J having continuous ECG waves,
    Output: Ɲ for each and every wave
1.  procedure OptiFog
2.    initialize $RT_o = 0$, $MU_o = 0$
3.  $T_s \leftarrow$ getTimeStamp();
4.  for i← 1, 2, ……. n do
```

```
5.      $T_{ijs} \leftarrow$ getFactors( $J_i$ , $S_i$ );
6.      $O_{ijr} \leftarrow$ receiveFactors($Jc_i$ , $S_i$);
        $T_{ijr} \leftarrow$ get($O_{ijr}$);$MU_i \leftarrow$ get($O_{ijr}$);
        $CU_i \leftarrow$ get($O_{ijr}$);$NC_i \leftarrow$ get($O_{ijr}$);
7.      $RT_i \leftarrow T_{ijs}$ - $T_{ijr}$
8.      $RT_o$ =$RT_o$ + $RT_i$
9.      $MU_o$ =$MU_o$ + $MU_i$
10.   **end for**
11.  $\boldsymbol{\alpha}_i \leftarrow 1, \forall i \in [1,2,...n]$.
12.  $\psi_i \leftarrow 0, \forall i \in [1,2,...n]$.
13.  j = n+1, i=1
14.  **while  j<=N do**
15.  calculate($Ͼ_i$, $Ł_i$, $\mu_i$);
16.  $\psi_{old} \leftarrow \psi_i$
17.  $\psi_i = 3Ͼ_i + 2Ł_i + 1\mu_i$
18.    **if**$\psi_i > \psi_{old}$**then**
19.  $\boldsymbol{\alpha}_i = \boldsymbol{\alpha}_i + 1$;
20.      assignTask($\boldsymbol{\alpha}_i$, i);
21.    **end if**
22.  **if**$\psi_i < \psi_{old}$**then**
23.  $\boldsymbol{\alpha}_i = \boldsymbol{\alpha}_i - 1$;
24.    assignTask($\boldsymbol{\alpha}_i$, i);
25.  **end if**
26.  **if** $\boldsymbol{\alpha}_I >= 0$ **then**
27.    j = j + $\boldsymbol{\alpha}_i$
28.  **end if**
29.  i = i + 1
30.  **if** i > n **then**
31.    i = 1
32.  **end if**
33.  **end while**
34.    $T_r \leftarrow$ getTimeStamp();
35.    $P_{OptiFog} \leftarrow T_{ir}$ - $T_{is}$
36.  **end procedure**
```

```
37.  Procedure assignTask( $\alpha$
     , i)
38.  if $\alpha_i$> 0 then
39.     $T_{is}$←getFactors( $J_\alpha$ , $S_i$ );
 40.    $O_{ir}$←receiveFactors($J_{\alpha c}$ , $S_i$);
           $T_{ir}$← get($O_{ijr}$);
           $MU_i$← get($O_{ijr}$);
           $CU_i$← get($O_{ijr}$);
           $NC_i$← get($O_{ijr}$);
         $RT_i$←$T_{ir}$ - $T_{is}$
 41.    end if
 42.    end procedure
```

Figure 5.9 beneath depicts the results after executing the OptiFog algorithm.

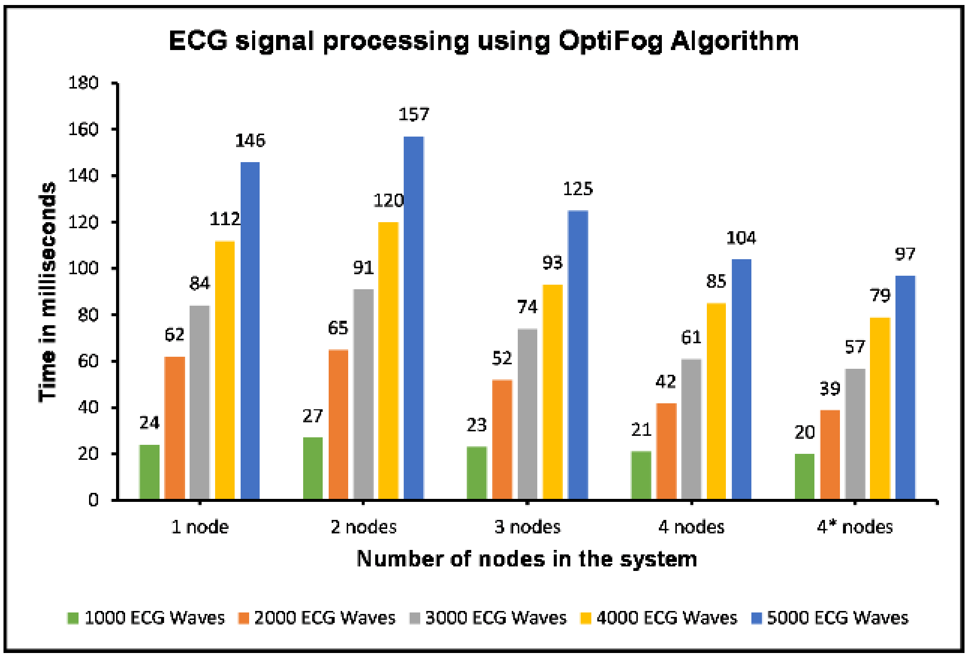

Figure 5.9 ECG signal processing using OptiFog Algorithm

Graph Interpretation:

- We can observe the impact of increase in the number of nodes.
- 4* nodes give fine performance results.
- The computation time required is lesser in comparison to ECG 8 wave and other techniques.

5.6 Testing OptiFog algorithm

For the proof of its rationality, the OptiFog algorithm is tested using 3 methods. The speedup factor is accounted in the first test scenario. In the second test case, the algorithm is executed for a greater number of ECG waves for observing the performance, in the subsequent test scenarios. For the third and final test scenario, the OptiFog algorithm is executed for dispy 10 wave case wherein the pre-loaded nodes within the cluster did not provide fine performance.

5.6.1 Test scenario 1: Speedup

In light of the experiments conducted until now in the Raspberry Pi clustering environment, each technique or parameter indicates a manner of improving performance. The Speedup factor can be utilized for comparing the job time J which is minimised due to these improvements.

$$Speedup_{overall} = \frac{Execution\ Time_{old}}{Execution\ Time_{new}} \text{..........................(8)}$$

Now, the other techniques are compared with the performance time of the dispy technique with double ECG waves and this time is also considered as reference time. Additionally, table 13 displays the step by step speedup betwixt other techniques. It indicates performance improvement with the considered series of algorithms. 14.4536 is the highest speedup offered by the OptiFog in 4* Nodes system. The performance is sped up by 1.1546 in consideration with 8 waves in 4* Nodes system.

Table 13: Speedup factors for 5000 ECG waves for 4 nodes and 4* Nodes system

Technique used (q)	**Speedup w.r.t Benchmark**		**Speedup w.r.t (q-1) technique**	
	4 Nodes system	4* Nodes system	4 Nodes system	4* Nodes system
Double waves	1	1	-	-
Tetra Waves	5.9118	5.8417	5.9118	5.8417
Octa Waves	11.256	12.5178	1.904	2.1428
OptiFog Algo.	13.529	14.4536	1.2019	1.1546

5.6.2 Test scenario 2

The suggested OptiFog algorithm is evaluated on a larger number of ECG waves. The system is tested against the OptiFog algorithm and the normal dispy system with 8 waves. For observing the performance in the worst case, the system is kept under a loaded scenario. 5000, 7500, and 10000 number of ECG waves are considered. Figures 5.10 and 5.11 display the results received by the dispy system and OptiFog algorithm respectively.

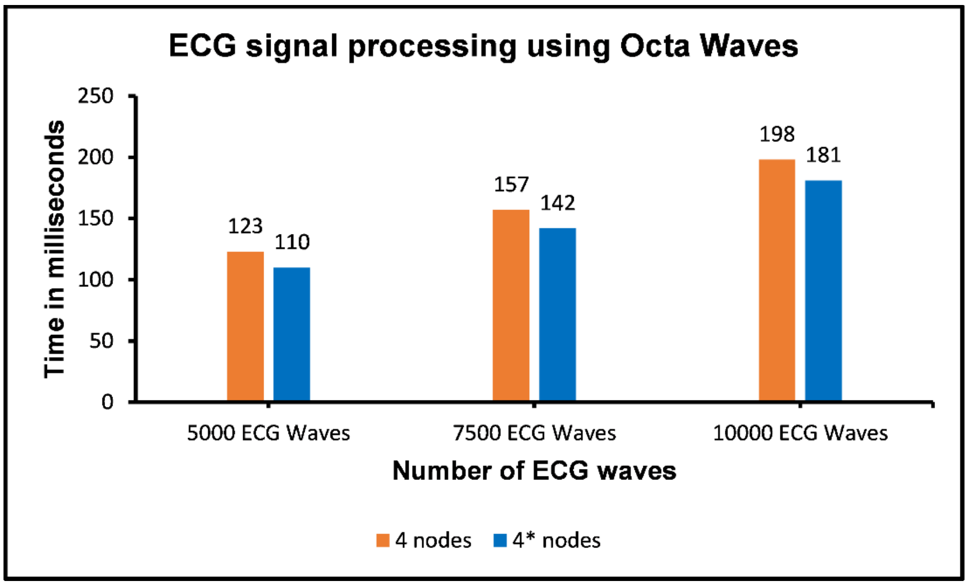

Figure 5.10 ECG signal processing by dispy system

From the above mentioned figures 5.10 and 5.11, we see that the OptiFog algorithm performs better than the dispy system and other algorithms with respect to computation. For larger jobs, OptiFog gives excellent results. With respect to performance, the impact of 4* nodes is very observable in this test scenario. The speedup factor for 4* Nodes system is 1.183 for 10000 ECG waves, which was 1.1546 for 5000 waves, both using OptiFog.

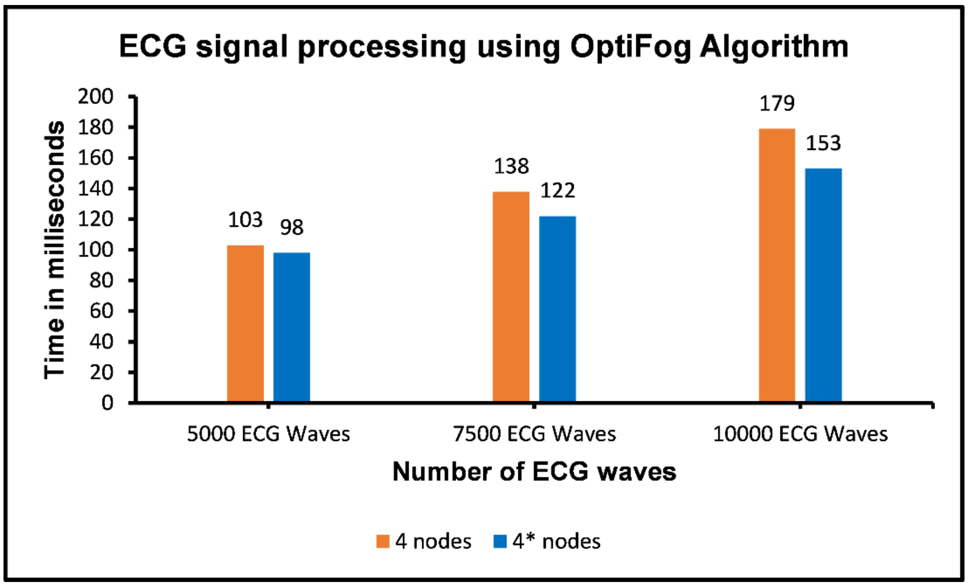

Figure 5.11 ECG signal processing by OptiFog Algorithm

This affirms that for larger number of waves the Speedup factor is enhanced. Figure 5.12 shows the conclusive improvement percentage. As the health care processing load increases The OptiFog does better as the load of healthcare processing increases. Further, its computational performance concerning dispy system also increases.

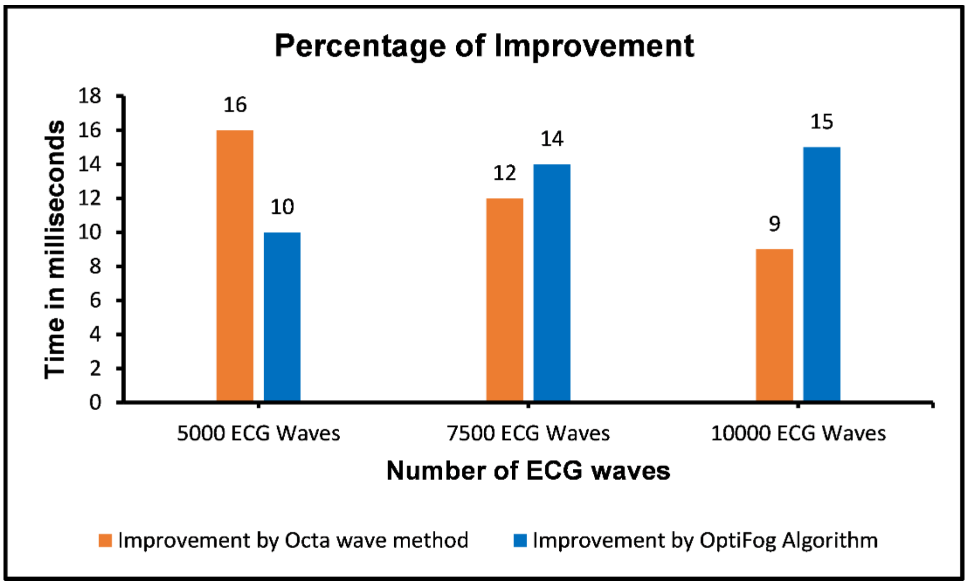

Figure 5.12 Percentage of improvement in the given test case

5.6.3 Test scenario 3

The system where the 10 waves ECG signal was processed using dispy was taken into consideration in this scenario, and we observed a decline in performance. Here, the network overhead seen is lesser because there are a number of waves which are deca. However, the deca wave loads cannot be managed by the loaded nodes as they utilize greater time than anticipated. The same case is tested with OptiFog wherein the job size based on individual node capability status is determined on by utilizing the impact ψ value. The result of this case is depicted beneath wherein OptiFog algorithm is executed with deca waves.

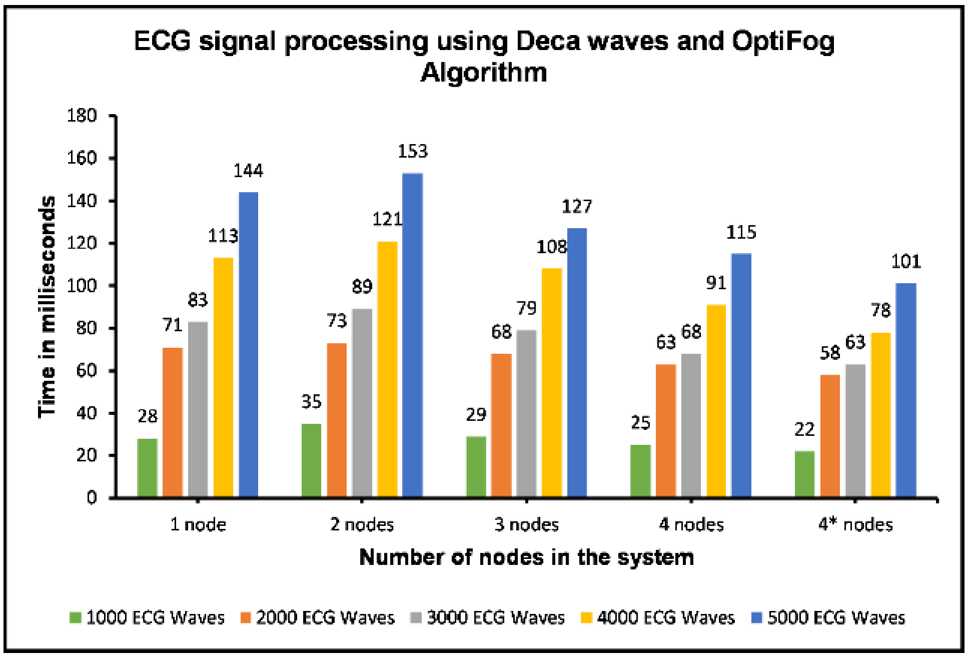

Figure 5.13 Running Deca waves using OptiFog Algorithm

Graph Interpretation:

- On account of existing preload and job load, we observe no enhancement for 1 node system compared to the dispy 10 waves graph.
- Due to the calculation overhead of Ͼ, Ł, μ and ψ the performance is declining in 2 Nodes system.
- On account of variation in job size and job allotting for available nodes in the 3 Nodes system, we observe the results to be better than 1 Node and 2 Nodes systems.

- We observe better performance for 4 Nodes system in comparison with 1 Node, 2 Nodes and 3 Nodes system.
- Due to OptiFog's ability in determining good impact factors each time and its ability to assign more tasks in sub job for 4* system, this system performs better than all the others.

OptiFog uses Distributed Computing to strengthen itself, and it is real-time processing algorithm expected in [120]. What makes it unique compared to the work done in this field is the usage of Heterogeneous computing and taking into consideration the worst scenario.

Concluding Remark

Analysis of ECG in real-time is a health care application which is time-critical. Any form of lag in this can prove to be life-threatening for patients. Although, Fog Computing faces dearth in terms of computation power [14], it has the ability to do this with minimised transmission delays. Raspberry Pi cluster is proposed for minimised delay in computation. For scalability and easy deployment in Distributed Computing, a good tool for utilizing in Pi cluster is Dispy. The size of the allotted sub-job should be adequate for obtaining fine performance from the dispy system. The overheads and master node iterations rely on the size of the sub-job, which can impact the performance of the system at higher levels. In terms of computation, each software and hardware parameter has an important role. CPU usage, response time, memory and number of cores utilized are the 4 parameters considered in this system and each has an impact on computation. The number of free cores and the CPU usage had a high effect while memory and response time had less effect on system performance for the current system. After taking into account, these effects and their effect levels, the OptiFog algorithm is designed with respect to varied factors and priorities. For assessing a node's processing capability, a fine measure is the impact factor. The OptiFog algorithm performs reasonably well for ECG health care data by using a Raspberry Pi cluster. As the OptiFog algorithm is planned for the worst scenarios, it shall always perform better. It is capable of determining and allotting optimal size of jobs for varied nodes in an environment that is heterogeneous. The performance of the OptiFog algorithm will also eventually rise compared to dispy systems and the test scenarios exhibited account for its performance. Thus, in the Heterogeneous Raspberry Pi clustering environment in Fog Computing, the OptiFog algorithm possesses the ability to yield better computations.