**American Journal of Mathematical and Computer Modelling**
2017; 2(2): 60-75
http://www.sciencepublishinggroup.com/j/ajmcm
doi: 10.11648/j.ajmcm.20170202.13

# Dynamic Load Balancing Using Periodically Load Collection with Past Experience Policy on Linux Cluster System

**Sharada Santosh Patil, Arpita Nirbhay Gopal**

MCA, Dept. Sinhgad Institute of Business Administration and Research, Kondhwa, Pune, Maharashtra, India

**Email address:**
sharada_jadhao@yahoo.com (S. S. Patil), arpit.gopl.@gmail.com (A. N. Gopal)

**To cite this article:**
Sharada Santosh Patil, Arpita Nirbhay Gopal. Dynamic Load Balancing Using Periodically Load Collection with Past Experience Policy on Linux Cluster System. *American Journal of Mathematical and Computer Modelling.* Vol. 2, No. 2, 2017, pp. 60-75.
doi: 10.11648/j.ajmcm.20170202.13

**Received**: October 21, 2016; **Accepted**: January 9, 2017; **Published**: March 9, 2017

**Abstract:** Fast execution of the applications achieved through parallel execution of the processes. This is very easily achieved by high performance cluster (HPC) through concurrent processing with the help of its compute nodes. The HPC cluster provides super computing power using execution of dynamic load balancing algorithm on compute nodes of the clusters. The main objective of dynamic load balancing algorithm is to distribute even workload among the compute nodes for increasing overall efficiency of the clustered system. The logic of dynamic load balancing algorithm needs parallel programming. The parallel programming on the HPC cluster can achieve through massage passing interface in C programming. The MPI library plays very important role to build new load balancing algorithm. The workload on a HPC cluster system can be highly variable, increasing the difficulty of balancing the load across its compute nodes. This paper proposes new idea of existing dynamic load balancing algorithm, by mixing centralized and decentralized approach which is implemented on Rock cluster and maximum time it gives the better performance. This paper also gives comparison between previous dynamic load balancing algorithm and new dynamic load balancing algorithm.

**Keywords:** MPI, Parallel Programming, HPC Clusters, DLBA ARPLCLB, ARPLCPELB

## 1. Introduction

High-performance clusters are implemented primarily to provide increased performance by splitting a computational task across many different nodes in the cluster, and are most commonly used in scientific computing. Such clusters commonly run custom programs which have been designed to exploit the parallelism available on HPC clusters. Many such programs use libraries such as MPI which are specially designed for writing scientific applications for HPC computers. (Michel Daydé, Jack Dongarra [2005]) [21] (G. Bums and R. Daoud, MPI Cubix - [1994]) [22].

Most of the HPC clusters consist of server and nodes. The server is responsible for distribution of the internet services to all other nodes. Other nodes are not directly connected with the internet. Hence this HPC cluster system is more secure. (Michel Daydé, Jack Dongarra – [2005]) [21].

The research is more challenging because of the various factors involved in implementing a load balancing algorithm in clustered system. Some of these influencing factors are the parallel workload, presence of any sequential and/or interactive jobs, native operating system, node hardware, network interface, network, and communication software. The main objective of load balancing algorithm is to speed up the system and enhance super computing power within the clustered system. There are two main types of performing load balancing – static load balancing and dynamic load balancing. (Paul Werstein, Hailing Situ and Zhiyi Huang [2006]) [9].

### *1.1. Static Load Balancing*

Static load balancing algorithm uses two renowned static policies are mentioned below.

(1). Load-dependent static policy.

(2). Speed–weighted random splitting policy (ParimahMohammadpour, Mohsen Sharifi, Ali Paikan-2008 ) [8].

The simulator designed in C language re sult is given below;

The overall output of algorithms is collected together, and comparative bar chart is drawn which is given in following Figure 1. These algorithms are local coscheduling algorithm, demand coschedulingalgorithmand dynamic load balancing algorithm The related comparison bar chart is given below:

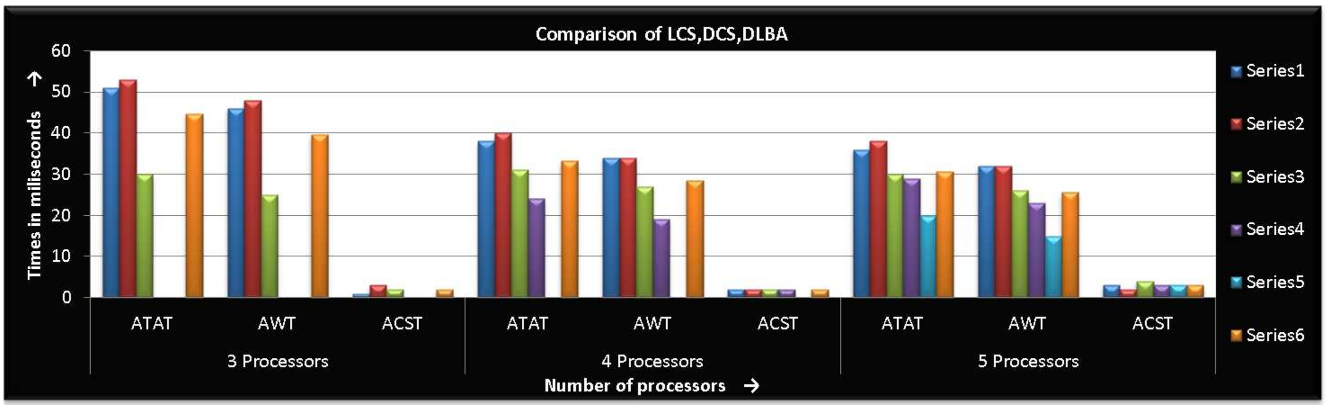


***Figure 1.*** *Comparison of coscheduling algorithm with DLBA using parameters.*

In above algorithms, it is proved that the overall performance of the dynamic load balancing algorithm is very good as compare to static coscheduling algorithms.

### *1.2. Dynamic Load Balancing Policies*

The dynamic load balancer distributes workload among the processors at run time. They have following policies (ParimahMohammadpour, Mohsen Sharifi, Ali Paikan-2008 ) [8].

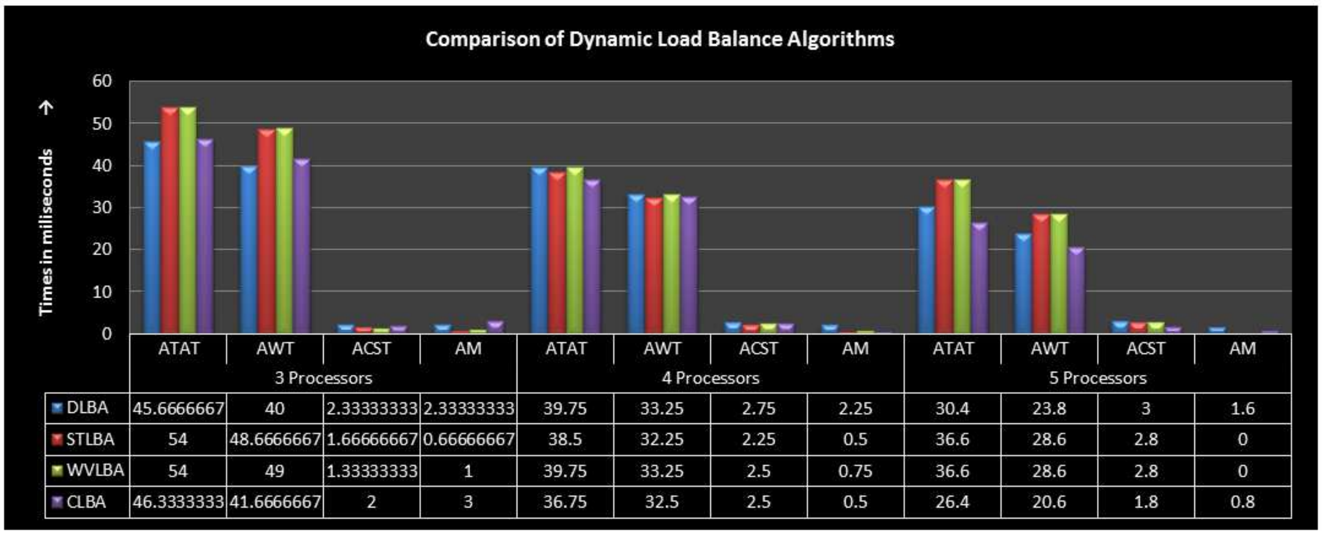
(1). Periodic policies

(2). Demand-driven policies

(3). State-change-driven policies

The simulator designed in C language re sult is given below;

The overall output of algorithms is collected together, and comparison bar chart is drawn which is given in following Figure 2. Theses algorithms are local coscheduling algorithm (discussed in section 4.2.1), demand coscheduling algorithm (discussed in section 4.2.2) and dynamic load balancing algorithm (discussed in section 4.2.3). The related comparison bar chart is given below:



| | 3 Processors | | | | 4 Processors | | | | 5 Processors | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ATAT | AWT | ACST | AM | ATAT | AWT | ACST | AM | ATAT | AWT | ACST | AM |
| 🟦 DLBA | 45.6666667 | 40 | 2.33333333 | 2.33333333 | 39.75 | 33.25 | 2.75 | 2.25 | 30.4 | 23.8 | 3 | 1.6 |
| 🟥 STLBA | 54 | 48.6666667 | 1.66666667 | 0.66666667 | 38.5 | 32.25 | 2.25 | 0.5 | 36.6 | 28.6 | 2.8 | 0 |
| 🟩 WVLBA | 54 | 49 | 1.33333333 | 1 | 39.75 | 33.25 | 2.5 | 0.75 | 36.6 | 28.6 | 2.8 | 0 |
| 🟪 CLBA | 46.3333333 | 41.6666667 | 2 | 3 | 36.75 | 32.5 | 2.5 | 0.5 | 26.4 | 20.6 | 1.8 | 0.8 |

***Figure 2.*** *Comparison of CLBA, STLBA, WVLBA, DLBA using Parameters.*

In above algorithms, it is proved that the overall performance of the dynamic load balancing algorithm is very good as compare to other dynamic load balancing algorithms.

The above diagram shows that STLBA has better performance bit as it follows a non pre-emptive centralized approach. However, DLBA gives poor result for the total migration time because it is fully dynamic pre-emptive scheduling algorithm, but gives good result for the average waiting time and average turnaround time of each process which is reduced.

## 2. The Message Passing Interface (MPI)

Message Passing Interface (MPI) is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. The standard defines the syntax and semantics of a core of library routines useful to a maximum size of users writing portable message-passing programs in Fortran 77 or the C programming language. According to R. Butler and E. Lusk P4 [2] is a third-generation parallel programming library, including both message passing and shared-memory components, portable to a great many parallel computing environments, including heterogeneous networks. Chameleon Written by W. D. Gropp and B. Smith [3] (Erik D. Demaine, Ian Foster, CarlKesselman, and Marc Snir [2001]) [4] (Hau Yee Sit Kei Shiu Ho Hong Va Leong Robert W. P. Luk Lai Kuen Ho [2004]) [18] (William Gropp, Rusty Lusk, Rob Ross, and Rajiv Thakur [2005]).

## 3. Authority Ring Periodically Load Collection for Load Balancing Algorithm (ARPLCLB)

The dynamic load balancing algorithm mainly removes the bottle necks presented by the static co-scheduling approach thus making the cluster co-scheduling scalable. But it presents a larger communication overhead as compared to static load balancing algorithm because dynamic load balancing algorithm performs process migration. In a centralized load balancing algorithm, load is distributed uniformly among the processors. The only disadvantage is maximum time of the central processor is wasted in load balancing rather than process execution. Hence performance of the server decreases. A decentralized load balancing algorithm decision of load distribution is taken by all the node hence, each node has load of other nodes and communication overhead increases tremendously. (Janhavi B, Sunil Surve, Sapna Prabhu-2010) [5].

In order to balance the load uniformly over a cluster system, one has to choose a mix of centralized and decentralized approach. (Janhavi B, Sunil Surve, Sapna Prabhu-2010) [5].

As said in above discussion, this algorithms mix two approaches - centralized and decentralized. The authority packet is circulated among the compute nodes. Whenever system is completely imbalanced, any lowly loaded processor can pick up this authority packet and get authority to become master node. Master node is responsible to balance the system. Every compute node can broadcast load to others at certain period such that they can evaluate their current state to find whether the system is balanced or imbalanced. Hence the name of this algorithm is Authority Ring Periodically Load Collection for Load Balancing Algorithm in short it is (ARPLCLB).

This section explains overall procedure, different policies used in the algorithm, Data structure used to build algorithm, and parallel algorithm.(Sharada Santosh Patil, ArpitaN. Gopal. [23].

Overall Procedure

The Overall Procedure of this algorithm is given below;

Step 1: After completion of every ring period, every processor passes or broadcasts information packet to all processor which consists of;

1. Current status of the node
2. Current load of the node with load factor

Step 2: Every processor can store the current information as well as past information of all the processors.

Step 3: Every processor collects authority packet from previous processor and circulate it to next processor.

Step 4: When any Idle node or Low load node get authority packet then immediately it take the charge of master node and performs following activities;

1. Create workload Distribution table using following process criteria;
   a. It chooses newly arrived processes. (That means new born Processes)
   b. It chooses processes which needs 80 % time for execution.
2. Create Order packets according to Workload Distribution table
3. Send order packet of all node to that appropriate node.
4. After load distribution send authority packet to next node.

Step 5: As soon as any node gets order packet they should follow the order of order packet to perform process migration.

Step 6: Master node again starts authority ring means authority packet is circulated to each node of the LAN one by one again.

Step 7: Repeats Steps 1 to 7 till cluster is not shut down (Sharada Santosh Patil, Arpita N. Gopal. -2013)[23].

## 4. The Idea of New Research

As it has been said before, The Authority Ring Periodically Load Collection for load balancing algorithm (ARPLCLB) uses centralized approach as well as decentralized approach. But it many a times it shows poor result. (SharadaSantoshPatil, Arpita N. Gopal.-2013)[23].

The advantages and disadvantages of the algorithm are given below;

*Advantages of Algorithm 1 (ARPLC):*

1. It dynamically distributes load and migrate processes from heavily loaded processes to idle or low loaded or normal loaded CPU successfully.
2. Mix centralized with decentralized approach for process migration.

*Disadvantages of Algorithm 1 (ARPLC):*

1. Too much communication overhead due to load collection at each iterations of the ring.
2. Master node always gives order and other has to follow it without any decentralized logic.
3. Process migration is very high.

The *major disadvantages* of Authority Ring Periodically Load Collection Algorithm (ARPLC).

1. Past Experience is not considered.
2. Nature of process means type of instructions are not considered.

3. CPU could not find self state.

4. Reselection Policy is not used means All nodes blindly follow the order of master node.

5. Load collection can be used all to all load passing policies, means, each node can pass their load to all other nodes.

6. Too much communication overhead.

Above disadvantages are very serious, hence there is urgent need to improve above algorithm. So that new dynamic load balancing algorithm need to use past experience of all process executed on the cluster. The most suitable name of this algorithm is Authority Ring Periodically Load Collection with Past experience for load balancing Algorithm which discussed in next section.

## 5. (ARPLCPELB)

This algorithm is again similar to ARPLCLB [23] but it increases period of load collection from other compute nodes. This algorithm considers past experience of process hence it reduces communication overhead and improves performance.

*Overall Procedure of Algorithm 2*

The overall procedure of the algorithm is given below;

Step 1: After some period every processors are passes or broadcast information packet to all processor which consists of;

1. Current status of the node
2. Current load of the node with load factor
3. Past experience of the processes.

Step 2: Every processor can store the current information as well as past information of all the processor.

Step 3: Every processor collect authority packet from previous processor and circulate it to next processor.

Step 4: When any idle node or Low load node get authority packet then immediately it takes the charge of master node and performs following activities;

1. Create workload Distribution table using following process criteria;

a. It chooses newly arrived processes. (That means new born Processes)

b. It chooses processes which needs 80 % time for execution with their past experience.

2. Create Order packets according to Workload Distribution table

3. Send order packet of all node to that appropriate node.

4. After load distribution send authority packet to next node.

Step 5: As soon as any node gets order packet they should follow the order of order packet to perform process migration.

Step 6: Master node again starts authority ring means authority packet is circulated to each node of the LAN one by one again.

Step 7: Repeats Steps 1 to 7 till cluster is not shut down

## 6. Policies Used in ARPLCLB Algorithm

Following Policies used in proposed dynamic load balancing algorithm.

### 6.1. Load Information Policy

Load information serves as one of the most fundamental elements in the load balancing process. Every dynamic load balancing algorithm is based on some type of load information. According to this algorithm, each load of cluster system has some current state. These CPU states can be idle, lowly loaded, normal or heavily loaded CPU.

1. The CPU state can be idle state, if ready queue is empty and it is not executing any process and hence 100 % memory is available.

$$\sum_{i=0}^{Q_{total}} P_i \approx 0 \,.\&.\, Memory_{utilization} \approx 0 \leftrightarrow \textbf{Equation} \; ❶$$

2. The CPU state can be lowly loaded, if total number of processes < (less than) LOW_LOAD_THRESHOLD_VALUE (i.e. L) * size of queue (Qs) and more than 75 % memory is available and 10% of total current processes are past processes.

$$\sum_{i=0}^{Qtotal} Pi \le L*Qs \,\&\, 75\% \le MEMfree \,\&\, \sum_{i=0}^{Qtotal} Pi \le \sum_{i=0}^{Qtotal} Ppei*0.1$$

3. The CPU state can be normal loaded if Total Number of processes < (less than) NORMAL_LOAD_THRESHOLD_VALUE * Size of Queue and 25 % to 75% memory is available and 30% of total current processes are past processes.

$$\sum_{i=0}^{Qtotal} Pi \le N*Qs \,\&\, 25\% \le MEMfree \le 75\% \,\&\, \sum_{i=0}^{Qtotal} Pi \le \sum_{i=0}^{Qtotal} Ppei*0.3$$

4. The CPU state can be heavily loaded if Total Number of processes > (greater than) NORMAL_LOAD_THRESHOLD_VALUE * Size of Queue and less than 25% memory is available and 70% of total current processes are past processes.

$$\sum_{i=0}^{Qtotal} Pi \le N*Qs \,\&\, 25\% > MEMfree \,\&\, \sum_{i=0}^{Qtotal} Pi \le \sum_{i=0}^{Qtotal} Ppei*0.7$$

5. The system balance depends on following criteria;

- When all nodes are heavily loaded then system can be called as heavily balanced system.
- When heavily loaded nodes are 1% to 85% and remaining are lowly loaded or idle or normal processors then system is imbalanced and need process migration
- When no node is heavily loaded and may be idle or lowly loaded or normal loaded then system is called slightly balanced or slightly imbalanced system, and it does not require any process migration.

### 6.2. Information Exchange Policies of ARPLCPELB

This information exchange policy depends on how node can be exchange load information with others. This algorithm uses periodic policy means it exchanges this load information after every certain number of time slot. This information policy is executed by all nodes.

### *6.3. Process Transfer Policies of ARPLCPELB*

Process can be transfer from heavily loaded processor to idle or lowly loaded or normally loaded processor. This policy is executed on master node. This policy can be mentioned as follows;

1. System is heavily balanced if all CPU in the clusters are heavily loaded then it execute delay of 1000 mille second such that all CPU can execute their load to get relief from authority token ring.
2. When system is slightly balanced or slightly imbalanced, then system is in normal condition
3. When system is completely imbalanced then this algorithm decides decision of process migration.
4. For process transfer activity, it calculates the ideal load of each processor as follows;

$$Ideal\_load = \frac{Total\_System\_load}{Total\_Computenode\_cluster}$$

5. It transfers total ideal load processes from heavily loaded processor to lightly loaded processor

### *6.4. Selection Policies of ARPLCPELB*

A selection policy decides which process is selected for transfer that means process migration. The chosen process could be a new process which has not started, that means, new born process or an old process which is already starting its execution. If it is old process then it should satisfy following condition.

If (rbt/bt*100>=80) Process is selected for migration.

Remaining_Burst_Time/Burst_Time*100>=90?:Process _Migration

This selection policy is executed by master node.

### *6.5. Location Policies of ARPLCPELB*

Location policy decides selected processes for migration is migrated to which CPU For this activity, it select ideal CPU to migrate processes from heavily loaded CPU to lightly loaded CPU using following steps.

1. Select heavy loaded CPU
2. Select first idle CPU

a. If found go to 3 else b

b. Select first low loaded CPU

i. If found go to step 3 else ii

ii. Select first normal loaded CPU

3. Select process for migration to selected CPU
4. Update load of that CPU
5. Repeat 3 and 4 till Load of CPU < ideal load of the system

This location policy is executed by master node.

## 7. Parallel Sub Algorithms of (ARPLCPELB)

This algorithm ARPLCLB is divided in to three parallel sub algorithms, that are;

1. Authority token ring with Periodically Load collection with Past Experience algorithm(ARPLCPE)
2. Load Distribution With Order Packet with Past Experience Algorithm(LDOP)
3. Process Migration with state logic Algorithm(PMSL)

### *7.1. Authority Token Ring with Periodically Load Collection Algorithm (ARPLC)*

This algorithm authority packet is moved around the logical ring of the compute nodes of the cluster. This algorithm circulates authority tokens between the processors such that they can choose their new master node when cluster system is imbalanced. This algorithm collects load information periodically, where at the time of load collection it also collect past experience of each process. The master node CPU can be responsible to start authorityring, in this it passes authority packet to next nearby neighbor. When system is imbalanced then any lowly loaded or idle node picks up this authority packet to become new master node. This node conveys this information to all using new master indication packet. This is explained in following algorithm 2.1.

**Algorithm 6.4:** Authority token Ring With Periodically Load collection Algorithm With Past Experience(ARPLCPE)

**Input** : total_cpu,cupid,next,prev,maser_node.

**Types of Packets:** AuthorityPacket, LoadPacket, MasterLoadPacket.

**Variables During Processing**:

i, slot=0,flag=0; idel=0,low=0, normal=0,heavy=0.

**Output** :New_master_node_ID

**Constant in the algorithm:**

Qsize, LOW_LOAD_THREASHOLD_VALUE, NORMAL_LOAD_THREASHOLD_VALUE, PERIOD_OF_LOAD _COLLECTION

**Procedure**:

```
Step 1: Initiallize all required variables
            Initiallize Parallel programming With MPI
            If  start==0        Then
                  Initialize authority Packet;
            End if
            prev = cpuid-1; next = cpuid+1;
            if cpuid == 0       Then
                  prev = total_cpu - 1;
            End if
            if cpuid == (total_cpu - 1)            Then
                  next = 0;
            End if
Step 2: Repeate following steps 3 to step 5
Step 3: if  cpuid==master_node  AND  flag==0
       Then
            Initialize authority Packet;
            Send athority_pkt to next cpu node with message tag tag1
            flag=1;
            Go to step 2
       Else
            Go to Step 4
       End if
Step 4:if  cpuid!=master_node   AND  flag==0
       Then
            Receive pkt from prev cpu node with message tag1
            If pkt==new master indication packet?
            Then
                  flag=2;
                  If next!=pkt[1]means master node id
                  Send athority_pkt to next cpu node with message tag1
                  Go to step 2
            Else
                  Go to step 4.1
            End if
      Else
            Go to step 5
      End if
Step 4.1:if  pkt == periodically_load_collection_packet?
       Then
            Send periodically_load_collection_packet to next node
            Initiallize load Packet and masternode packet
            Broadcast load_pkt to all and collects other load_pkt in master_load_pkt
            Go to step 2
       else
            Go to Step 4.2
       End if
Step 4.2:If athority_pkt completes one round ?  Then
            idel=0;low=0;normal=0;heavy=0;
            for(i=0;i<total_cpu;i++)
            begin
                  if  load(CPU_i)=0?  Then
                        idel++;
                  Else
                  If load(CPU_i) < LOW_LOAD_THREASHOLD_VALUE*Qsize Then
                        low++;
                  Else
```

```
                                        If load(CPU_i) < NORMAL_LOAD_THREASHOLD_VALUE*QsizeThen
                                                normal++;
                                        Else
                                                heavy++;
                                End for
                                Go to Step 4.3
Step 4.3: if heavy==total_cpu
                Then
                                Execute delay(1000);
                Else
                If(((idel>0)||(low>0)||(normal>0))&&(heavy>0))       Then
                                if load(cupid)<= LOW_LOAD_THREASHOLD_VALUE*Qsize Then
                                        flag=2;
                                Else
                                        Go to step 4.4
                                End if
                Else
                                 flag=0;
                                Go to step 4.4
                End if
Step 4.4:If flag ==0             Then
                                Send athority_pkt to next cpu node with message tag tag1
                                Go to step 2
                Else
                  If flag ==2
                  Then
                                  athority_pkt[0]=-1;athority_pkt[1]=cpuid;
                                  Send master_indication_pkt to next cpu node with tag1
                                  Go to step 2
                  End if
                End if
Step 5:if((cpuid==master_node)&&(flag==1))          Then
                                Receive pkt from prev cpu node with message tag1
                                /*Prepare periodically load all gather*/
                                If(athority_pkt[athority_pkt_size-1]==-1)                Then
                                    If(slot== PERIOD_OF_LOAD _COLLECTION)  Then
                                        athority_pkt[0]==-2;
                                        athority_pkt[athority_pkt_size-1]=-2;
                                        Send periodically_load_collection_packet to next node
                                        Go to step 2
                                    Else
                                        slot++;
                                Else
                                   If pkt==new master indication packet?     Then
                                         flag=2;
                                         If next!=pkt[1]means master node id
                                         Send master_indication_pkt to next cpu node with tag1
                                         Go to step 2
                                   else
                                        Go to step 5.1
                                   End if
                                End if
            Else
                                Go to step 5.5
            End if
Step 5.1:if pkt == periodically_load_collection_packet?            Then
                                Initiallize load Packet and masternode packet
                                Broadcast load_pkt to all and collects other load_pkt in master_load_pkt
                                athority_pkt[0]=master_node;slot=0;
```

```
                    Send athority_pkt to next cpu node with message tag1
                    Go to step 2
            Else
                    flag=1;  athority_pkt[1]++;
                    athority_pkt[athority_pkt_size-1]=-1;
                    Goto Step 5.2
            End if
Step 5.2:If  athority_pkt completes one round            Then
                    idel=0;low=0;normal=0;heavy=0;
                    for(i=0;i<total_cpu;i++)
                    begin
                              if  load(CPU_i)=0?            Then
                                        idel++;
                              else end if
                              if   load(CPU_i) < LOW_LOAD_THREASHOLD_VALUE*Qsize?        Then
                                        low++;
                              else end if
                              if  load(CPU_i) < NORMAL_LOAD_THREASHOLD_VALUE*Qsize ?
                              Then
                                        normal++;
                              else
                                        heavy++;
                              end if
                    End for
                    Go to Step 5.3
Step 5.3:If  heavy==total_cpu
             Then
                    Execute delay(1000);
             Else
               If  ((idel>0)||(low>0)||(normal>0))   AND   (heavy>0) ?          Then
                    If  load(cupid)<= LOW_LOAD_THREASHOLD_VALUE*Qsize
                    Then      flag=2;
                    Else
                              Go to step 5.4
                    End if
               Else
                     flag=1;
                    Go to step 5.4
             End if
           End if
Step 5.4:If flag ==1            Then
                    Send athority_pkt to next cpu node with message tag tag1
                    Go to step 2
             Else
                 If flag ==2
                 Then
                      athority_pkt[0]=-1;athority_pkt[1]=cpuid;
                      Send master_indication_pkt to next cpu node with tag1
                      Go to step 2
                 End if
            End if
Step 5.5 : if  flag==2     Then
                              Store  master_node in history;
                              master_node=athority_pkt[1];
                              Go to Step 6
               End if
Step 6 : Call Load Distribution Algorithm With Periodically Load collection Algorithm With Past Experience
```

### *7.2. Load Distribution with Order Packetusing Past Experience Algorithm (LDOPPE)*

When cluster system is imbalanced and new master node is decided by ARPLCPE parallel sub algorithm, then this new master node has load information of each node. Hence it decides load distribution using load information as well as past experience of the compute node of cluster. During decision it uses distribution policy, process selection and location policy, accordingly creates order packets, and distributes or broadcasts this order packet to each node, and calls process migration algorithm which follows order of master node blindly. Maximum part of this LDOPPE algorithm is executed on master node and very minimum part of algorithm is executed on other compute node. This is explained in following algorithm.

**Algorithm 6.8:** Load Distribution With Periodically Load Collection Algorithm With Past Experience (LDPLCPE)

**Input** **: Load master Packet**.

**Types of Packets :**OrderPacket

**Types of Array :**actual_load, cpu_status, vcpu_status, virtual_load

**Variables During Processing**:

i,j,k,op,total_load,flag=0; idel=0,low=0,

normal=0,heavy=0,balance_factor,dest,transfer_flag,src_ldm_addr, src,

src_ord_addr,CPU_state.;

**Output** **:Order Packet**

**Constant in the algorithm:**

Qsize, LOW_LOAD_THREASHOLD_VALUE, NORMAL_LOAD_THREASHOLD_VALUE,

**Procedure**:

```
Step 1: Initiallize all required variables
                i=0;j=0;k=-1;op=0;total_load=0;
                Opl=QSize*total_cpu
                Allocate opl memory to order_pkt Initialize it
Step 2://All cpu calculate self state
        If load($CPU_{id}$)=0? Then
                CPU_state=0;
        Else
        If load($CPU_{id}$) < LOW_LOAD_THREASHOLD_VALUE*Qsize Then
                CPU_state=1;
        Else
        If load($CPU_{id}$) < NORMAL_LOAD_THREASHOLD_VALUE*QsizeThen
                CPU_state=2;
        Else
                CPU_state=3;
        End if
Step 3: if cpuid==master_node
         Then
                Allocate total_cpu memory to actual_load,cpu_status,vcpu_status,virtual_load
                //Master CPU Calculate actual load of the cpu
                for(i=0,k=-1;i<lplm;i++)
                begin
                        If ( i%(QSize*7(i.e. recordsize))==0) Then
        k++;actual_load[k]=0;j=0;virtual_load[k]=0;
                        End if
                        If (load_pkt_master[i]!=-1) Then
                                actual_load[k]++;total_load++;i+=9;
                        Else
                                i=(k+1)*(7*QSize)-1;
                        End if
                End for
                // Master CPU Calculate total status of the cpu
                for(i=0;i<total_cpu;i++)
                Begin
                        If(actual_load[i]==0)Then
                                cpu_status[i]=0;
                        Else
                        if(actual_load[i]<LOW_LOAD_THREASHOLD_VALUE *QSize)
        Then
                                cpu_status[i]=1;
                        Else
                        If(actual_load[i]<NORMAL_LOAD_THREASHOLD_VALUE*QSize)
                        Then
                                cpu_status[i]=2;
                        Else
                                cpu_status[i]=3;
                        End If
                        vcpu_status[i]=cpu_status[i];
                End for
                Balance_Factor=total_load/total_cpu;
                Go To step 3.1
         Else
                Go to Step 4
         End if
Step 3.1: Repeat Steps 3.2 to 3.
Step 3.2: i=0;heavy=-1;
         /*Select heavy loaded node */
                for(i=0;i<total_cpu;i++)
                Begin
                        if((cpu_status[i]==3)&&(vcpu_status[i]==3))
                        Then
                                heavy=i;
                                Go to Step 3.3;
                        End if
                End for
                If(heavy==-1)
                Then
                        Go to Step 4;
                End if
Step 3.2:  /*Select idle or low loaded or normal loaded node */
                idle=-1;/*Select Idel Node */
                for(i=0;i<total_cpu;i++)
                Begin
                        if(vcpu_status[i]==0)
                        Then
                                idle=i;
                                Go to Step 3.3;
                        End if
                End for
                if(idle==-1)
                Then
                        low=-1;/*Select low Node */
                        for(i=0;i<total_cpu;i++)
                        Begin
                                If(vcpu_status[i]==1)
                                Then
                                        low=i;
                                        Go to Step 3.3;
                                End if
                        End for
                        If(low==-1)
                        Then
                                normal=-1;/*Select low Node */
                                for(i=0;i<total_cpu;i++)
                                Begin
                                        if(vcpu_status[i]==2)
                                        Then    normal=i;
                                                Go to Step 3.3;
                                        End If
                                End for
                        End if
                End if
                Go to Step 3.3
Step 3.3: if(((idle==-1)&&(low==-1))&&(normal==-1))
         Then
                Go to Step 4;
         Else
                src=heavy;
                If(idle!=-1)
                Then
                        dest=idle;
                Else
                If(low!=-1)
                Then
                        dest=low;
                Else
                        dest=normal;
                End if
         End if
         Go to Step 3.4;
Step 3.4:lbf=(actual_load[src]+actual_load[dest])/2;
         /* Load Distribution Logic */
          src_ldm_addr=src*7*QSize;
          src_ord_addr=src* QSize;
          for(i=src_ldm_addr;i<(src_ldm_addr+7* QSize);i+=7)
          Begin
                if(((load_pkt_master[i+2]/load_pkt_master[i+1])*100) > 90)
                Then
                        order_pkt[src_ord_addr]=dest;
                        src_ord_addr++;
                        virtual_load[dest]++;virtual_load[src]--;
         transfer_flag++;
                Else
                If((((load_pkt_master[i+2]/load_pkt_master[i+1])*100)>80)
                        &&((load_pkt_master[i+6]-load_pkt_master[i+3])
         >load_pkt_master[i+1]))
                Then
                        order_pkt[src_ord_addr]=dest;
                        src_ord_addr++;
                        virtual_load[dest]++;virtual_load[src]--;
                        transfer_flag++;
                Else
                if((((load_pkt_master[i+2]/load_pkt_master[i+1])*100)>60)
                        &&((load_pkt_master[i+5]-load_pkt_master[i+3])
                >load_pkt_master[i+1]))
                Then
                        order_pkt[src_ord_addr]=dest;
                        src_ord_addr++;  transfer_flag++;
```

```
                    virtual_load[dest]++;virtual_load[src]--;
                    src_ord_addr++;
             Else
                    order_pkt[src_ord_addr]=-1;
                    src_ord_addr++;
             End if
             if((actual_load[dest]+virtual_load[dest])>=lbf)
             Then
                    Go to **Step 3.5**;
             End if
             if((virtual_load[src]+actual_load[src])>=(virtual_load[dest]+actual_load[dest]))
             Then
                    Go to **Step 3.5**;
             End if
        End for
        Go to Step 3.5
Step 3.5: /*change the status of each cpu according to its virtual load*/
         for(i=0;i<total_cpu;i++)
        Begin
             if((actual_load[i]+virtual_load)==0)
             Then
                    vcpu_status[i]=0;
             Else
              if((actual_load[i]+virtual_load[i])<LOW_LOAD_THREASHOLD_VALUE  *
       QSize)
             Then
                    vcpu_status[i]=1;
             Else
       if((actual_load[i]+virtual_load[i]) <NORMAL_LOAD_THREASHOLD_VALUE                  *QSize)
             Then
                    vcpu_status[i]=2;
             Else
                    vcpu_status[i]=3;
             End if
        End for
        Go to Step 3.6;
Step 3.6: old_dest=dest;
        If( (  ( (idle==-1)&&(low==-1) )  && (normal==-1) )&&(transfer_flag>0))
        Then
             Go to Step 4;
        End if
        Go to Step 3;
Step 4: /* Broad cast order packet to each node*/
       Wait until all cpu come to this point through Barrier(MPI_COMM_WORLD);
       Broad cast all order packet to all cpus in the cluster
Step 5: Call Process Migration with CPU State logic Algorithm
```

**Note: 1 Step 1, Step 2, Step 4 and Step 5 are executed by all CPU node**
**2 Step 3 and its sub states are only executed by the Master node**

### *7.3. Process Migration Using State Logic of Algorithm (PM)*

Once order packet is distributed by the new master node among all other compute nodes of the cluster then other compute nodes are automatically divided in to heavy load CPU group and idle or low load CPU group. And heavy loaded CPU transfers their own load to lightly loaded CPU according to state logic. This is explained in figure 3. The load is transferred using two types of the packets, that means process control block packet whose size is fixed and then it transfers actual process to destination CPU such that it can be easily start its remaining execution on new processor.

State logic means when any CPU has order to migrate process and its current state is not heavily loaded it is normal or lowly loaded then this CPU not migrates any process.

When process migration is over it again execute the

ARPLC parallel sub algorithm to monitor system imbalance and distribute authority packet among the compute nodes

**Algorithm 6.9:** Process migration with CPU State Logic (**PMSL**)

**Input** **: OrderPacket**.

**Types of Packets:** PCBPacket, ProcessPacket

**Variables During Processing**:

pcbl,pl,mpl,i,r;

**Output** **:Process Migration from old CPU to new CPU**

**Algorithms Used:**

Process_Migration(ProcessId,src_cpu,dest_cpu)

Aadnya_Palan()

**Procedure**: Process_Migration (pid,src_cpu,dest_cpu)

```
Step 1: Initiallize all required variables
                Allocate memory ot pcb
                Initiallize pcb according to pid
Step 2://Check with source cpu
        If(cpuid==src_cpu)
        Then
                pl=allot process_contrl_blokof(pid);
                Allocate Memory to process
                Load process
                Send pcb_pkt to dest_cpu with pcb_msg_tag
                Send process_pkt to dest_cpu with process_msg_tag
        Else
        If(cpuid==dest_cpu)
        Then
                Receive pkt from prev cpu node with pcb_msg_tag
                pl=lenth mentioned in pcb i.e. pcb_pkt[1];
                Allocate memory to process;
                Receive pkt from prev cpu node with process_msg_tag
                Load process
                Execute process
        End if
Step 3:Return
```

**Procedure**: Aadnya_Palan()

```
Step 1:If(CPU_State==3)/*Heavy Loaded Processor
        Then
                Go to Step 2
        Else
                Go to Step 3
        End If
Step 2://Check Order Packet
        for(i=0;i<QSize;i++)
        Begin
                if(order_pkt[cpuid*QSize+i]<0)
                Then
                        Continue with for loop;
                End if
                //Call Process Migration Algorithm
                Process_Migration(i,cpuid,order_pkt[cpuid*QSize+i]);
        End for
Step 4:// Idel Low or Normal Loaded Processor*/
```

```
        for(r=0;r<total_cpu;r++)
        Begin
                if(r==cupid)
                Then
                        Continue with for loop;
                End if
                for(i=0;i<QSize;i++)
                Begin
                        If(order_pkt[r*QSize+i]<0)
                        Then
                                Continue with for loop;
                        End if
                        If(order_pkt[r*QSize+i]==cpuid)
                        Then
                                //Call Process Migration Algorithm
                                Process_Migration(i,r,rank);
                        End if
                End for
        End For/* Destination Loop*/
State 5: Return back to run Authority Ring Periodically load collection algorithm PE
```

**Note: 1 Step 1, Step 2, Step 4 and Step 5 are executed by all CPU node**
**2 Step 3 and its sub states are only executed by the Master node**

***Table 1.*** *Performance of Algorithm1 (ARPLCLB).*

| Iteration of Outer Loop | Total Process Migration | Total Number of Rings | New Master | Old master |
|---|---|---|---|---|
| 1 | 9 | 2 | 2 | 0 |
| 2 | 9 | 1 | 2 | 2 |
| 3 | 8 | 6 | 2 | 2 |
| 4 | 8 | 2 | 0 | 2 |
| 5 | 8 | 1 | 0 | 0 |
| 6 | 8 | 4 | 3 | 0 |
| 7 | 9 | 4 | 0 | 3 |
| 8 | 8 | 4 | 3 | 0 |
| 9 | 8 | 1 | 0 | 3 |
| 10 | 8 | 4 | 3 | 0 |

## 8. Performance of the Authority Ring with Periodically Load Collection Algorithm

For evaluating the performance of the above algorithm we implement algorithm run it on Rock cluster on centos operating system then it produces some output in data file. These data files are too lengthy hence only some result are given, following figure shows screen shot of the same during the execution. following figure shows screen shot of the same during the execution.

The authority rings continues, means system is currently in balanced state. When system is in imbalanced stage, then it performs process migration. The graph of outer loop iteration against process migration is given in figure 6;

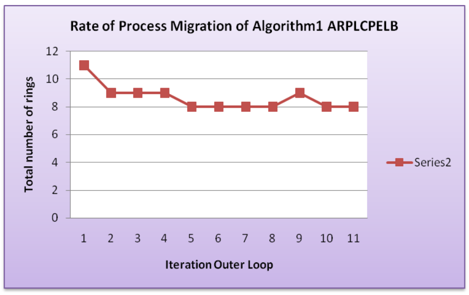


***Figure 3.*** *Performance of ARPLCPELB using Process migration.*

As result shows, this algorithm migrates maximum number of processes in each number of iterations of the loop. Hence Maximum time of the CPU is wasting to perform process migration rather than process execution. Hence it is degradation of the system.

The graph shows total authority rings verses iteration of outer loop. When total authority rings are more means system is currently in balanced state.

The graph shows total authority rings verses iteration of outer loop. When total authority rings are more means system is currently in balanced state.

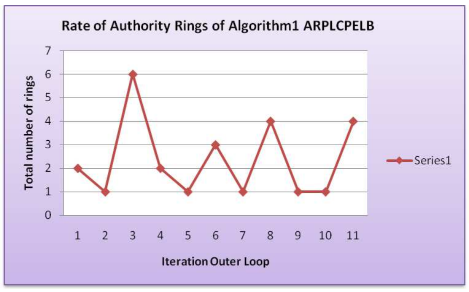


***Figure 4.*** *Performance of Algorithm 1 Using Total Rings (ARPLCPELB).*

The result of above graph states that every iteration of the ring migrates to too much processes. And this algorithm uses periodically load collection policy, hence, it uses too much communication overhead. The advantages and disadvantages of the algorithm are given below;

*Advantages of Algorithm 2 (ARPLCPELB):*

1. It dynamically distributes load and migrate processes from heavily loaded processes to idle or low loaded or normal loaded CPU successfully.
2. It gives better performance than ARPLCLB
3. Process selection policy uses past experience for process migration.
4. Its communication overhead is less as compare to ARPLCLB

*Disadvantages of Algorithm 2 (ARPLCPELB):*

1. Too much communication overhead due to periodically load collection policy.
2. Master node always gives order and other has to follow it without any logic on process level.
3. Process migration is very high.

## 9. Comparison of ARPLCLB and ARPLCPELB based on Algorithmic Logic

The nature of software always depends on logic of the software. Hence the detailed comparison is based on logic used in algorithm. The similarities of these algorithms are given below;

### *9.1. Similarities of Algorithm Based on Logic*

1 All proposed Algorithms mix centralize and decentralize logic together

2 All Algorithms have three main modules

3 All Algorithm works in similar manner such that each node passes authority token when system is imbalanced any low loaded or idle compute node can pick up this authority token and inform all other nodes that new master node is changed with the help of master indication packet

4. Master node distributes load among the nodes using Load distribution algorithm

5 Process migration activity follows the order of order packet

6All compute nodes has 4types of states that are idle or lowly loaded, normal loaded or highly loaded CPU

### *9.2. Dissimilarities of Algorithm Based on Logic*

The differences of these algorithms are given below;

***Table 2.*** *Comparison of Algorithm Based on Logic.*

| Sr. No | ARPLCLB | ARPLCPELB |
|---|---|---|
| 1 | Past Experience is not considered | Past Experience is considered. that means Actual execution time required for the process with turnaround time and wait time used for process selection for the migration. Hence logic of process selection is improved. |
| 2 | Nature of process means type of instructions are not considered | Nature of process means type of instructions are not considered |
| 3 | CPU could not find self state | CPU can finds self state Hence logic of order followers in process migration are improved |
| 4 | Reselection Policy is not used means All nodes blindly follow the order of master node | State logic is not used means All nodes blindly follow the order of master node |

## 10. Comparison of ARPLCLB and ARPLCPELB Based on Policies

The policies used in the algorithms always play very significant role in point of view of performance of the algorithm. The similarities of both algorithm based on policies used within algorithms are given below;

### *10.1. Similarities of Algorithm Based on Policies*

1. Both Algorithms information exchange policy is same that is periodically load collection policy.
2. Both algorithms process transfer policy is same. i.e. When system is completely imbalanced then it decides decision of process migration.
3. Both algorithms location policy is same. i.e. When system is completely imbalanced then it decides about process migration from heavy loaded CPU to low loaded idle or normal CPU. This policy is implemented in all the algorithms.

### *10.2. Dissimilarities of Algorithm Based on Policies*

Hence detailed comparison based on policies used in the algorithm are explained in the following table 3

***Table 3.*** *Comparison of Algorithm Based on Policies.*

| Policy | ARPLCLB | ARPLCPELB |
|---|---|---|
| Load Estimation Policy | Memory utilization and total number of processes in ready queues are considered but Past Experience is not considered | Memory utilization and total number of processes in ready queues are considered with Past Experience like turn-around time, actual time etc. |
| Selection Policy | For Process selection, only memory and remaining burst time is considered. | For Process selection, only memory and remaining burst time is considered with past experience. |

## 11. Conclusion and Future Enhancement

This algorithm balance the load uniformly over a HPC cluster system, The performance of this algorithm gives better result many a time but due to heavy communication overhead and heavy process migration, affect the performance. Hence this previous algorithm is redesigned such that it considers past experience during the load distribution decision but at the time of process migration current state of the CPU is considered for migration decision. Hence proposed algorithm gives better result than previous result. The future work is extended to reduce communication overhead between the compute nodes.

This work is extended to remove all disadvantages of this proposed algorithm so as to improve its performance. As well as policies used in this algorithm is also improved. In future, this work can be extended to develop new dynamic load balancing algorithm to modify dynamic decentralized approach so as to reduce the communication overhead as well as to reduce migration time and also make it scalable.

## References

[1] Bernd F reisleben Dieter Hartmann ThiloKielmann [1997] “Parallel Raytracing A Case Study on Partitioning and Scheduling on Workstation Clusters” 1997 Thirtieth Annual Hawwaii International Conference on System Sciences.

[2] Blaise Barney, (1994) Livermore Computing, MPI Web pages at Argonne National Laboratory http://www-unix.mcs.anl.gov/mpi "Using MPI", Gropp, Lusk and Skjellum. MIT Press

[3] Erik D. Demaine, Ian Foster, CarlKesselman, and Marc Snir [2001] “Generalized Communicators in the Message Passing Interface” 2001 IEEE transactions on parallel and distributed systems pages from 610 to 616.

[4] Hau Yee Sit Kei Shiu Ho Hong Va Leong Robert W. P. Luk Lai Kuen Ho [2004] “An Adaptive Clustering Approach to Dynamic Load balancing” 2004 IEEE 7th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN’04).

[5] JanhaviB, SunilSurve, Sapna Prabhu-2010 “Comparison of load balancing algorithms in a Grid” 2010 International Conference on Data Storage and Data Engineering Pages from 20 to 23.

[6] M. Snir, SW. Otto, S. Huss-Lederman, D. W. Walker and J. Dongarra,(1996) MPI: The Complete Reference (MIT Press, Cambridge, MA, 1995). 828 W. Gropp et al./Parallel Computing 22 (1996) 789-828.

[7] Marta Beltr´an and Antonio Guzm´an [2008] “Designing load balancing algorithms capable of dealing with workload variability” 2008 International Symposium on Parallel and Distributed Computing Pages from 107 to 114.

[8] ParimahMohammadpour, Mohsen Sharifi, Ali Paikan,[2008] “A Self-Training Algorithm for Load Balancing in Cluster Computing”, 2008 IEEE Fourth International Conference on Networked Computing and Advanced Information Management, Pages from 104 to 110.

[9] Paul Werstein, Hailing Situ and Zhiyi Huang [2006] “Load Balancing in a Cluster Computer” Proceedings of the Seventh International Conference on Parallel and Distributed Computing, Applications and Technologies.

[10] SharadaPatil, DrArpita Gopal,[2012], MsPratibhaMandave “Parallel programming through Message Passing Interface to improving performance of clusters” – International Docteral Conference (ISSN 0974-0597) SIOM, WadgoanBudruk in Feb 2013.

[11] SharadaPatil, Arpita Gopal “Comparison of Cluster Scheduling Mechanism using Workload and System Parameters” 2011 ISSN 0974-0767 International journal of Computer Science and Application.

[12] SharadaPatil, Arpita Gopal “STUDY OF DYNAMIC LOAD BALANCING ALGORITHMS FOR LINUX CLUSTERED SYSTEM USING SIMULATOR” 2011 ISSN 0974-3588 International journal of Computer Applications in Engineering Technology and Sciences.

[13] SharadaPatil, DrArpita Gopal, [2011] “Study of Load Balancing ALgorithms” – National Conference on biztech 2011, Selected as a best paper in the conference, got first rank to the paper, DICER, Narhe, Pune in year March 2011.

[14] SharadaPatil, DrArpita Gopal, [2013] “Cluster Performance Evaluation using Load Balancing Algorithm” – INTERNATIONAL CONFERENCE ON INFORMATION COMMUNICATION AND EMBEDDED SYSTEMS ICICES 2013,978-1-4673-5788-3/13/$31.00©2013IEEE (ISBN 978-1-4673-5786-9) Chennai, India in year Feb 2013.

[15] SharadaPatil, DrArpita Gopal,[2012] “Need Of New Load Balancing Algorithms For Linux Clustered System” – International Conference on Computational techniques And Artificial intelligence (ICCTAI’2012) (ISBN 978-81-922428-5-9) Penang Maleshia in year Jan 2012.

[16] SharadaPatil, DrArpita Gopal,[2013] “Enhancing Performance of Business By Using Exctracted Supercomputing Power From Linux Cluster’s ” – International Conference on FDI 2013 (ISSN 0974-0597) SIOM, WadgoanBudruk in Jan 2013.

[17] Sun Nian, Liang Guangmin [2010] “Dynamic Load Balancing Algorithm for MPI Parallel Computing” 2010 Pages 95 to 99.

[18] William Gropp, Rusty Lusk, Rob Ross, and Rajiv Thakur [2005] “MPI Tutorials “ Retrieved from www.mcs.anl.gov/research/projects/mpi/tutorial Livermore Computing specific information:

[19] Yanyong Zhang, AnandSivasubramaniam, JoseÂ Moreira, and Hubertus Franke [2001] “Impact of Workload and System Parameters on Next Generation Cluster Scheduling Mechanisms” 2001 IEEE transactions on parallel and distributed systems Pages from 967 to 985.

[20] Yongzhi Zhu Jing GuoYanling Wang [2009] “Study on Dynamic Load Balancing Algorithm Based on MPICH” 2009 MPI_COMM_RANK: World Congress on Software Engineering. Pages from 103 to 107.

[21] Michel Daydé, Jack Dongarra (2005) “High Performance Computing for Computational Science - VECPAR 2004” ISBN 3-540-25424-2 pages 120-121.

[22] G. Bums and R. Daoud, MPI Cubix (1994) Collective POSIX I/O operations for MPI, Tech. Rept. OSC-TR- 1995- 10, Ohio Supercomputer Center, 1995.

[23] Sharada Santosh Patil, Arpita N. Gopal. Authority Ring Periodically Load Collection for Load Balancing of Cluster System. American Journal of Networks and Communications. Vol. 2, No. 5, 2013, pp. 133-139. doi: 10.11648/j.ajnc.20130205.13.