Website: ijetms.in Issue:1, Volume No.2, January 2018

Mixing of Centralized and Decentralized Methods Using Demand Driven Load Collection in Dynamic Load Balancing Algorithm in Cluster System

Dr. Sharada Santosh Patil

[1]Associate Professor MCA, Deptt. SIBAR Kondhwa, Pune, Maharashtra, INDIA

Email address:
sharada_jadhao@yahoo.com(S. S. Patil)

Abstract: The load balancing algorithm distributing homogeneous load among the cluster hence increases the speed of high performance clustered system due to its parallel computation capabilities because of its compute nodes. The most attractive point of load balancing algorithm is to distribute load of heavily loaded compute node among lightly loaded compute nodes during the execution, which is called as process migration. This process migration time can be saved using new method in this algorithm. Hence some policies are needed to consider at the time of load transfer decision as well as at the time of process migration. The construction of dynamic load balancing algorithm requires MPI instructions to achieve needs parallel programming. The parallel programming on the cluster can be execute using massage passing interface (MPI) or application programming interface (API). This paper uses only MPI library to build new load balancing algorithm. Due to very highly variable workload of a cluster system, the difficulty of load balancing is also increasing across its compute nodes. This paper proposes new approach of existing dynamic load balancing algorithm, which is implemented on Rock cluster and maximum time it gives the better performance. This algorithm uses demand driven load collection methods so its speed is increases. This paper focused and on comparison between previous dynamic load balancing algorithm and also gives performance of new dynamic load balancing algorithm.

Keywords: Concurrent Programming, MPI Library, HPC Clusters, DLBA ,ARPLCLB, ARPLCPELB, ARDLCLB ,LDOP,PM , PMDL

1. Introduction

Parallel CPUs are connected on mother board called as HPC cluster, which speed performance can be increased by splitting a computational task across various nodes in the cluster, which are most commonly used in scientific parallel computing. Generally such clusters executes custom programs. These custom programs have been designed to take advantage of the concurrent programming available on HPC clusters. The parallel programming environment can provided through Message passing interface i.e. MPI library **(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 accountable for distribution of the internet services to all other nodes. **(Michel Daydé, Jack Dongarra – [2005]) [21]**

Every load balancing algorithm has very well known objective is to speed up computer system and to increase super computing power within the clustered system. This load balancing algorithms can be divided in to two main types – static load balancing and dynamic load balancing. **(Paul Werstein, Hailing Situ and Zhiyi Huang [2006]) [9].**

The Static load balancing algorithms are always centralized and run on central CPU rather than decentralized approach, hence only centralized node in responsible for decision of load balancing. Where as dynamic load balancing algorithm distributes workload among the processors at run time. The information exchange policies are periodic policy or demand driven policy or state change driven policy, **(Parimah Mohammadpour, Mohsen Sharifi, Ali Paikan-2008) [8]**.

There are three main sources of load imbalance, they are application imbalance, workload imbalance and heterogeneity of hardware resources. **(Yongzhi Zhu Jing Guo Yanling Wang [2009])[20] (Marta Beltr´an and Antonio Guzm´an [2008]) [7]**

Load balancing in the application level thinks on reducing the turn around time or completion time of an application where as load balancing in the system level is thinks on maximizing the throughput or utilization rate of the nodes. **(Bernd F reisleben Dieter Hartmann Thilo Kielmann [1997])[1]**

These load balancing schemes can centralized or decentralized. When only central node collects the information of each node and performs the decision making based on overall knowledge of the system is called centralized load balancing. When all the nodes are taking the decision of load balancing based on its local knowledge then it is called as distributed load balancing scheme. **(Parimah Mohammadpour, Mohsen Sharifi, Ali Paikan-2008)(Janhavi B, Sunil Surve ,Sapna Prabhu-2010) [8][5]**

The load balancing algorithm can be supports non pre-emptive and pre-emptive.

When only new processes means new born processes ,(they are not started their execution yet) only are transfer from highly loaded processor to lightly loaded processor such a scheme is adopted in non pre-emptive load balancing scheme.

When any process may be running processes also , are transfer from highly loaded processor to lightly loaded processor such a scheme is adapted by pre-emptive load balancing scheme. **[17] (Sun Nian, Liang Guangmin [2010])[19](Yanyong Zhang, Anand Sivasubramaniam, JoseÂ Moreira, and Hubertus Franke [2001])[20](Yongzhi Zhu Jing Guo Yanling Wang [2009]).**

The researcher from academia and industries has designed portable message-passing system designed to function on a wide variety of parallel computers. It consists of the standard defines the syntax and semantics of a core of library functions useful to a large 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 concurrent programming library,which consisting both message passing and shared-memory components, portable to a parallel processing environments. Chameleon Written by **W.D. Gropp and B. Smith [3](Erik D. Demaine, Ian Foster,Carl Kesselman, 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])**

2. ARPLCLB and ARPLCPELB Algorithms

Load collection policy of these two dynamic load balancing algorithm is periodically load collection approach. The main disadvantage of these algorithms is maximum time of the central processor is wasted in load balancing rather than process execution. Hence performance of the server decreases. These algorithms also having a decentralized load balancing algorithm, so 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]**

The new idea of load balancing, 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]**

The ARPLCLB and ARPLCPELB algorithms mix two approaches - centralized and decentralized. According to these algorithms, the authority packet is circulated circularly between the CPU nodes as well as periodically each CPU is broad casting their nodes to all other nodes.

Each node is collecting load information and analyze about the system whether it is balanced or imbalanced.

Authority packet can assign authority to CPU to take decision of load balancing. The moment on which 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. Hence the name of this algorithm is Authority Ring Periodically Load Collection for Load Balancing Algorithm in short it is (ARPLCLB) **(Sharada Santosh Patil, Arpita N.Gopal.) [23] , (Sharada Santosh Patil, Arpita N.Gopal.) [24]**

3. The Idea of New Research

The centralized load balancing approach is mixed with decentralized load balancing approach in Authority Ring Periodically Load Collection for load balancing algorithm (ARPLCLB) but it many a times it shows poor result. **(Sharada Santosh Patil, Arpita N. Gopal. -2013)[23]**

The ***major disadvantages*** of Authority Ring Periodically Load Collection with past experience algorithm (ARPLCPE) are given below.

1. Too much communication overhead due to periodically load collection policy.
2. Master node always gives order and the other has to follow it without any logic.
3. Process migration is still high.
4. Migrated processes are again selected for migration.
5. Too much communication overhead due to periodic load collection.

These above mentioned disadvantages are very serious, the major reason behind these disadvantages are periodic load collection method , hence there is urgent need to change load collection policy which can improve above algorithm. So that new dynamic load balancing algorithm need to use demand driven load collection policy. The most suitable name of this algorithm is Authority Ring with demand load collection for load balancing Algorithm (ARDLCLB) which discussed in next section.

4. The ARDLCLB Algorithm:

This algorithm does not collect a load periodically using all to all logic. It uses demand driven techniques with all to one technique, hence it saves too much communication overhead. This algorithm also uses decentralized activity at the time of process migration. This section explains the overall procedure, different policies used in the algorithm, Data structures used to build algorithm, and parallel algorithm.

4.1. Overall Procedure

The Overall Procedure of this algorithm is given below;

Step 1: When the system is imbalanced then old master is changed to the new master and according to his demand every processor are passing or broadcast information packet to all processors which consists of:

1. Current status of the node
2. Current load of the node with load factor
3. Information about all the processes with their types (CPU bound,IO bound, Memory bound).
4. Past experience of the processes.

Step 2: Every processor has to maintain the current information as well as past information of all the processor with nature of instructions (CPU bound instruction or IO bound instructions) of that process.

Step 3: Every processor can pick up authority packet and maintain the current load information in the authority packet and circulate it to next processor.

Step 4: It checks if system is imbalanced if so and any idle node or lightly loaded node get authority packet then immediately it performs following activities:

1. Collects information of processors using load packets
2. It selects processes for migration from heavy loaded processor using following process criteria:

a. It chooses newly arrived processes. (i.e. new born processes)

b. It chooses processes which needs 80 % time for execution. This time can be calculated by past experience of the processes as well as nature of instructions used in that processes.

3. Create workload distribution table . and also creates order packets according to workload distribution table.
4. Send order packet of all nodes to that appropriate node.
5. After load distribution, it perform process migration.

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

Step 6: After the load balancing operation is over again master node starts authority ring using circulating of authority packet to each node of the cluster.

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

5. Policies Used in ARDLCLB Algorithm

The required policies used in proposed dynamic load balancing algorithm are explained below.

5.1. Load Information Policy

Load information is used as the most fundamental elements in the load balancing process, without which load balancing decision is impossible. According to this algorithm, there are 4 current states of every CPU. That are idle, lowly loaded, normal or heavily loaded CPU.

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

$$\sum_{i=0}^{Qtotal} Pi \;=\; 0\&100\% MEMfree$$

2. The lowly loaded CPU state can be defined as total number of processes (Pi) < (less than) LOW_LOAD_THRESHOLD_VALUE (L)* Size of Queue (Qs) and more than 75 % memory(MEM_{free}) is available and 80% of total current processes with past experience(P_{pei}) and Nature of the processes are CPU bound processes.

$$\sum_{i=0}^{Qtotal} Pi \le L*Qs \,\&\, MEMfree > 75\% \,\&\, \sum_{i=0}^{Qtotal} Pi \le \sum_{i=0}^{Qtotal} Ppei*0.6\,\&$$
$$((CPUinstr >= 25\%) AND (IOinstr + MEMinstr) <= 75)$$

3. The normal loaded CPU state can bedefined as total Number of processes (Pi) < (less than) NORMAL_LOAD_THRESHOLD_VALUE **(N)** * Size of Queue and 25 % to 75% memory is available(MEM_{free}) and 60% of total current processes with past experience(P_{pei}) and Nature of the processes are 50%IO(IO_{instr})+Memory(MEM_{instr}) and 50% CPU bound (CPU_{instr})processes.

$$\sum_{i=0}^{Qtotal} Pi \le N*Qs \,\&\, 25\% \le MEMfree \le 75\% \,\&\, \sum_{i=0}^{Qtotal} Pi \le \sum_{i=0}^{Qtotal} Ppei*0.6\,\&$$
$$((CPUinstr >= 50\%) AND (IOinstr + MEMinstr) <= 50)$$

4. The heavily loaded CPU state can be defined as total

number of processes > (greater than) NORMAL_LOAD_THRESHOLD_VALUE * Size of Queue and less than 25% memory is available (MEM_{free}) and 40% of total current processes with past processes (P_{pei}) and Nature of the processes with 75%IO(IO_{instr})+Memory(Mem_{instr}) and 25% CPU bound (CPU_{instr})processes.

$$\sum_{i=0}^{Qtotal} Pi \le N*Qs \,\&\, MEMfree < 25\% \,\&\, \sum_{i=0}^{Qtotal} Pi \le \sum_{i=0}^{Qtotal} Ppei*0.4 \,\&$$

$$((CPUinstr >= 25\%) AND (IOinstr + MEMinstr) <= 75)$$

- When hundred percent means all nodes are heavily loaded, then the clustered system can be called as heavily balanced system.
- When among the cluster system, heavily loaded nodes are 1% to 85% and the 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 the system is called slightly balanced or slightly imbalanced system which is not required any process migration.

This policy is executed by the entire node.

5.2. Information Exchange Policies of ARDLCLB

This information exchange policy depends on how node can exchange load information with others. This algorithm uses **demand load collection policy** means it exchange this load information when the system is imbalanced and new master node demanded the load of each node. This load is only collected by the master node, hence communication overhead reduces tremendously.

5.3. Process Transfer Policies of ARDLCLB

The process transfer policy is given below.

1. When the system is heavily balanced and all CPU in the clusters are heavily loaded then it executing delay of 1000 ms such that all CPUs can execute their load to get relief from authority token ring circulated among the clusters.
2. When the system is slightly imbalanced or slightly balanced then system is called as normal condition
3. When the system is completely imbalanced then this algorithm change master node and perform decision of process migration by the master node using centralized approach.
4. It calculates these Ideal load of each processor and translates all processes till ideal load of lowly loaded or idle or normally loaded CPU.;

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

5. When other node gets order then they can reselect processes which can be migrated to destination node without changing the format of order packet

5.4. Selection Policies of ARDLCLB

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 chosen process is the old process then it should satisfy following condition.

1. If (rbt/bt*100>=90) Process is selected for migration.

$$\frac{Remaining\_Burst\_Time}{Burst\_Time}*100>=90?:Process\_Migration$$

2 If(remaining burst time / burst time *100>=80)&&(actual avarage execution time- wait time > burst time) then process is selected for migration

$$(\frac{Remaining\_Burst\_Time}{(Burst\_Time)}*100>=80) and$$

$$((Actual\_Time-Wait\_time) > Burst\_time)?:Process\_Migration$$

3 If((remaining burst time / burst time *100>=60)&&(Total Number of execution !=0)&&(turnaround around time – wait time > burst time) && ((cpu_instr+mem_instr+io_instr)/io_instr>50) then process is selected for migration

$$(\frac{Remaining\_Burst\_Time}{(Burst\_Time)}*100>=60) and (\frac{CPUinstr+IOinstr+MEMinstr}{IOinstr}>50) and$$

$$((TurnArround\_Time-Wait\_time) > Burst\_time) and$$

$(TotalExecution > 0)?:Process\_Migration$

This selection policy is executed by master node.

5.5. Location Policies of ARPLCPELB

Location policy of the clusters decides selected processes for migration is migrated to end location of CPU. For this activity, it selects ideal or lowly loaded CPU to migrate processes from heavily loaded CPU . It uses 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.

6. Parallel Sub Algorithms of ARDLCLB)

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

1. Authority token ring with demand Load collection with Past Experience algorithm(ARDLCPE)
2. Load Collection With Order Packet Creation Algorithm (LCOP)
3. Process Migration with decentralized logic(PMDL)

6.1. Authority Token Ring with demand Load Collection Algorithm (ARDLC)

This algorithm uses similar logic regarding authority packet but it does not use periodic policy to collect load of the each nodes. It uses authority packet to convey current status of the node. Hence the system is balanced or imbalanced can be find out using authority packet. Whenever one ring is completed, means authority packet has load of each node then any idle node or lowly loaded node find out whether the system is balanced or imbalanced state.

If the system is in balanced state, then it continues authority ring otherwise it pickup authority packet and become new master node to demand load from each node.

For this purpose new master node converts authority packet into new master indication packet or load demand packet and circulate it to its neighbor using authority ring. Hence every node can know about their new master node as well as load demand, so as to stops authority ring and ready to send their individual load to new master node.

Here load factor of each node also considers past experience as well as also considers nature of the instructions. This is explained in following figure: 1

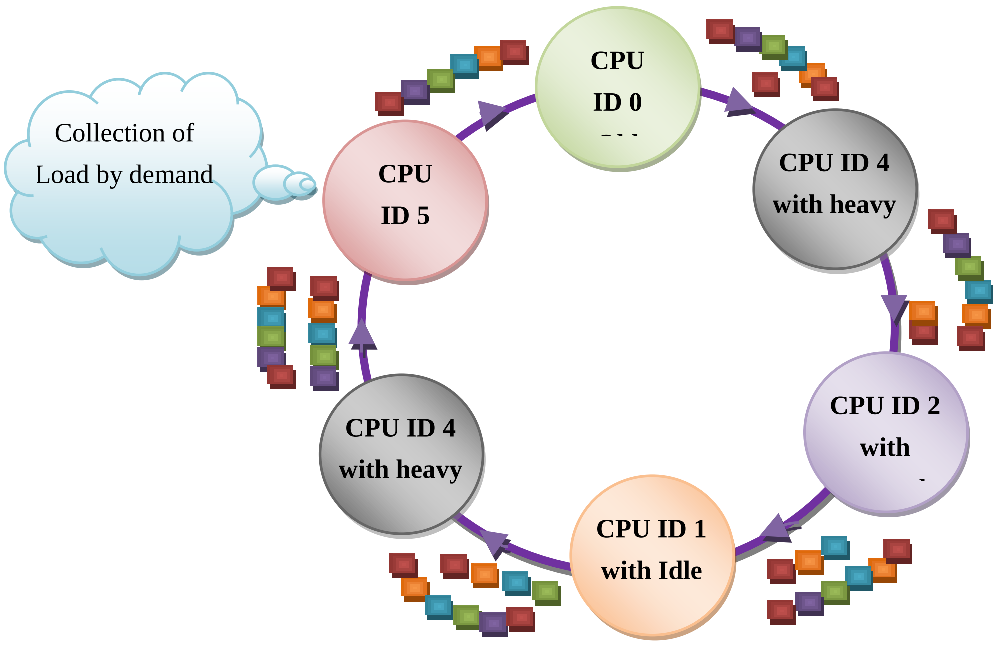

Figure 1: *Authority token Ring with Demand Load collection (ARDLCPE)*

The detailed algorithm of ARDLC is given below:

Algorithm 5.7: Authority token Ring With Demand Load Collection Algorithm (ARDLC)
Input **:** total_cpu,cupid,next,prev,maser_node.
Types of Packets **:** AuthorityPacket.
Variables During Processing:
i, 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.

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
                End if
            Else
                Go to step 4.1
            End if
        Else
            Go to step 5
        End if
Step 4.1: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*Qsize             Then
                    normal++;
                Else
                    heavy++;
            End for
            Go to Step 4.2
Step 4.2: 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 4.3
            End if
         Else
            flag=0; Go to step 4.3
         End if
Step 4.3: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) AND (flag==1))             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 ?
                Then
                    Send master_indication_pkt to next CPU node with tag1
                    Go to step 2
                End if  Else
                Go to step 5.1
            End if      Else
```

```
                Go to step 5.4
        End if
Step 5.1: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++;
                        end if
                        if  load(CPU_i) < NORMAL_LOAD_THREASHOLD_VALUE*Qsize ?
                        Then
                                normal++;
                        else
                                heavy++;
                        end if
                End for
                Go to Step 5.2
Step 5.2: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 5.3
                End if
           Else
                 flag=1;
                Go to step 5.3
          End if
        End if
Step 5.3: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 5.4 : 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 Demand  Load collection Algorithm
```

6.2. Load Collection with Order Packet creation (LCOP)

In previous both algorithms, all nodes collects load of other nodes periodically hence when system is imbalanced then master node not collects load information, it directly creates order packets for process migration.

But this load information may change due to some duration, hence this algorithm first collect latest new load information from each node (**Demand of** master node) and then decides load distribution and accordingly create order packets. This is explained in following figure 2.

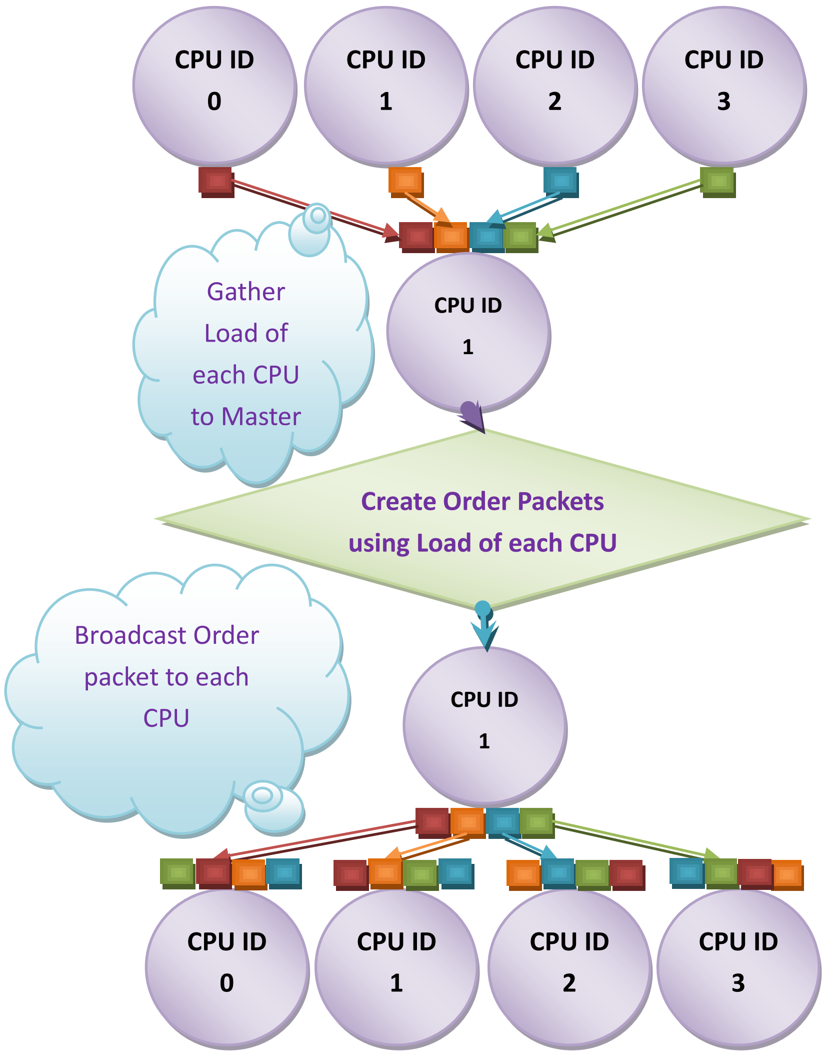
CPU ID 0
CPU ID 1
CPU ID 2
CPU ID 3
Gather Load of each CPU to Master
CPU ID 1
Create Order Packets using Load of each CPU
Broadcast Order packet to each CPU
CPU ID 1
CPU ID 0
CPU ID 1
CPU ID 2
CPU ID 3

figure 2. Load Collection With Order Packet (LCOP)

The detailed algorithm of LCOP is given below:

Algorithm 5.8: Load Collection With Order Packet Creation Algorithm (LCOP)

Input **: Load master Packet**.

Types of Packets : LoadPacket, MasterLoadPacket,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,lbf,CPU_state.;
```

Output **:Order Packet**

Constant in the algorithm:

```
                    Qsize,LOW_LOAD_THREASHOLD_VALUE,
                    NORMAL_LOAD_THREASHOLD_VALUE,
                     BALANCE_FACTOR
```

Procedure:

```
Step 1: Initiallize all required variables
                i=0;j=0;k=-1;op=0;total_load=0;
                Initialize load_pkt
                 If cupid==master_node
                Then
                    Initiallize masterload_pkt
                End if
                Gather all node loadpkt from all nodes and store in to masterload_pkt
                Opl=QSize*total_cpu
                Allocate opl memory to order_pkt and initialize order packet
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*Qsize
        Then
                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*10(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)*(10*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;
           End if
Step 3.4:
           lbf=(actual_load[src]+actual_load[dest])/2;
           /* Load Distribution Logic */
            src_ldm_addr=src*10*QSize;   src_ord_addr=src* QSize;
            for(i=src_ldm_addr;i<(src_ldm_addr+10* QSize);i+=10)
            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+9]-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]))
                              &&(  ((load_pkt_master[i+8]/(load_pkt_master[i+6]
                      +load_pkt_master[i+7]+load_pkt_master[i+8])*100)
         >50)))
                    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 reselection decentralize 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

6.3. Process Migration using decentralized logic of Algorithm (PMDL)

Previous both process migration algorithms do not implements reselection of process by the heavy loaded CPU hence it can migrate maximum time process to lightly loaded CPU. This process also execute the state logic as well as process reselection logic.

After accepting order packet, this algorithm divides CPU’s in to two groups, that is prcess sender group and process receiving group. Usually according to order packet process sender group contain heavily loaded processors and process receiving group contain lowly loaded or idle or normal processors.

But due to some time duration, some of the heaviely loaded processors are changing their state from heaviely loaded node to normal or lowly loaded node, then such a node not transfers their load, but they communicates regarding the same to respective waiting nodes. This logic is called as a **state logic**.

This algorithm also execute **decentralized process reselection logic.** According to this logic, when any heaviely loaded processor transfers processes to other nodes then it (i.e. heavily loaded node) finds current status of the process which is selected for the migration. Using that status, it finds outs its remaining burst time with the help of nature of instructions used in process as well as from past experience of the process. If its remaining burst time is less than 50 % then that process is not migrated otherwise it is selected for migration. Process migration factor is also considered in reselection.

Figure 3: Process Migration with Decentralized Logic (PCDL)

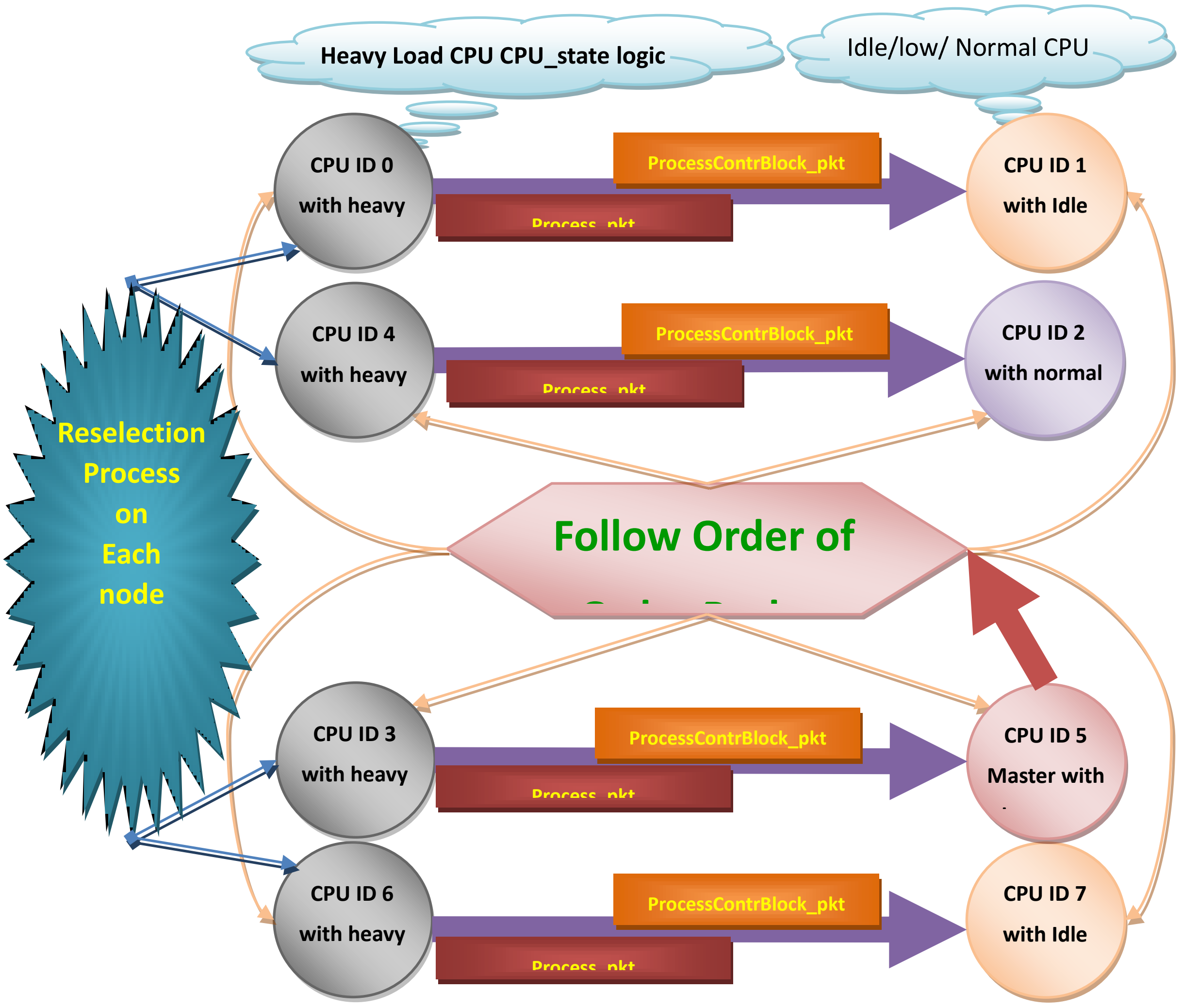

Figure 3: Process Migration with Decentralized Logic (PCDL)

The detailed algorithm of PMDL is given below:

Algorithm 5.9: Process migration with Decentralized Logic (**PMDL**)
Input : **OrderPacket**.
Types of Packets : PCBPacket, ProcessPacket
Types of Array :migrated_process
Variables During Processing:

```
                        Pcbl,pl,mpl,i,r,pn,
                        total_Migrated_Processes=0,
                        run,tm;
Output          :Process Migration from old CPU to new CPU
Constant used: MIGRATION_FACTOR
Algorithms Used:
        Process_Migration(ProcessId,src_cpu,dest_cpu)
        Reselection_Process()
        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
                If(pid<0)
                Then
                        pcb_pkt[1]=-1;
                        /*First send PCB msg*/
                        Send pcb_pkt to dest_cpu with pcb_msg_ tag
                Else
                        pl=allot process_contrl_blokof(pid);
                        Allocate Memory to process
                        Load process
                        Send pcb_pkt to dest_cpu with pcb_msg_tag
                        /* send pcb*/
                        Send process_pkt to dest_cpu with process_msg_tag
                        /*send process*/
                End if
        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];
                If(pl>=0)
                Then
                        Allocate memory to process;
                        Receive pkt from prev CPU node with process_msg_tag
                        Load process
                        Execute process
                End if
        End if
Step 3:Return
```

Procedure: Reselection_Process()

```
State 1:if(CPU_State==3)/*Heavy Loaded Processor*/
        Then
                Go to Step 2
        Else
                Return
        End if
Step 2: mpl=QSize;
        Allocate memory to migrated_process
Step 3:Change order of order packet
                for(i=0;i<QSize;i++)
                Begin
                        //Initiallize migrated packet
                        If(order_pkt[cpuid*QSize]<0)
                        Then
                                migrated_process[i]=-1;
```

```
                    continue with for loop;
                End if
                tm=total_Number_of_migration(i);
                run=remaining_run_pcb(i);
                If((run>50)&&(tm<MIGRATION_FACTOR))
                Then
                    migrated_process[i]=i;
                Else
                    migrated_process[i]=New_Process_ID(i);
                End if
                total_Migrated_Processes++;
            End for
Step 4: Return
```

Procedure: Aadnya_Palan()

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

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 Performance of the Authority Ring with Demand Load Collection Algorithm

This algorithm gives following result during the execution. The main difference between previous algorithms and this algorithm is that this algorithm is demand driven load collection algorithm. Hence, communication overhead is reduced rapidly. This algorithm also considers process migration factor and at the time of process migration, all heavily loaded CPU again reselects processes for migration so cost of migration is savedin following table1.

Table 1: *Performance of Algorithm3 (ARDLCLB)*

Iteration of Outer Loop	Total Process Migration	Total Number of Rings	New Master	Old master
1	2	1	2	0
2	1	2	2	2
3	1	13	3	2
4	1	3	3	3
5	1	2	0	3
6	1	3	3	0
7	1	1	0	3
8	1	4	0	0
9	2	2	0	0
10	1	3	2	0
11	1	2	0	2
12	1	1	2	0
13	1	8	1	2
14	1	1	2	1
15	1	1	2	2
16	1	1	0	2
17	1	2	3	0
18	1	2	1	3
19	1	2	1	1
20	1	1	1	1

Advantages of Algorithm 3 (ARDLCLB):

1. It dynamically distributes load and migrates processes from heavily loaded processes to idle or low loaded or normal loaded CPU successfully.
2. It gives better performance than algorithm 1 as well as algorithm 2 also.
3. Process selection policy uses past experience for process migration.
4. For process selection policy it uses nature of the process also.
5. Its communication overhead is less as compare to Algorithm 1and Algorithm 2
6. Communication overhead is very less as compared to previous algorithms because of the demand driven information policy.

8 Overall comparison of ARDLCLB with other algorithms

The overall performance of ARDLCLB is very good which is shown in following graphs:

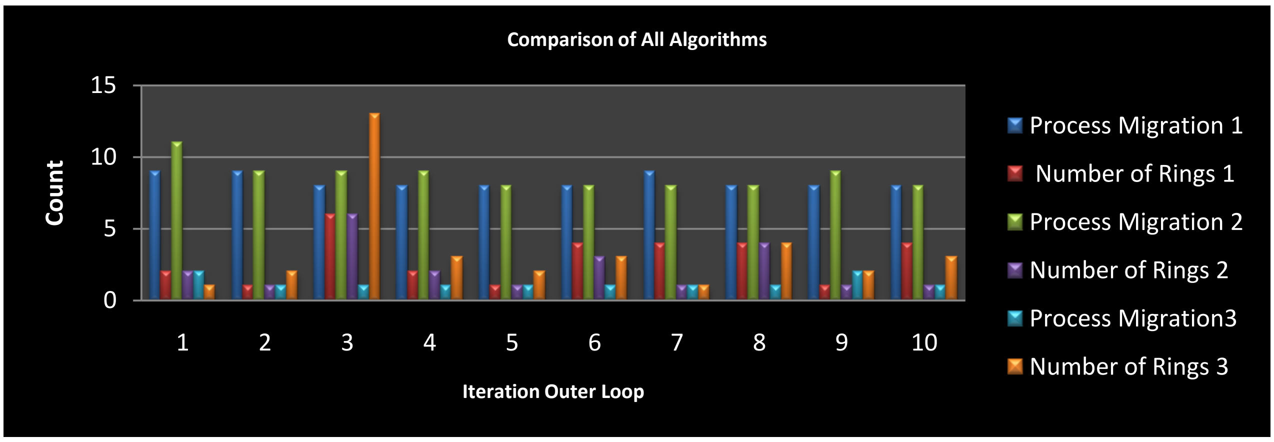

Figure 6.7 Comparison of proposed Algorithms (ARPLCLB, ARPLCPELB, ARDLC)

The result of the above bar chart is given below:

- In above chart, it is seen that process migration of first two algorithms is very high, but it is less in algorithm 3.
- The total rings of algorithm 3 are more as compare to algorithm 1 and algorithm 2. Hence it is proved that system is more balanced in algorithm 3.
- It gives very good results because of decentralized logic.
- It also controls communication overhead because of the demand driven policy of the algorithm.

The result of the above bar chart of figure 6.7 is clearly given below by using bar chart of process migration:

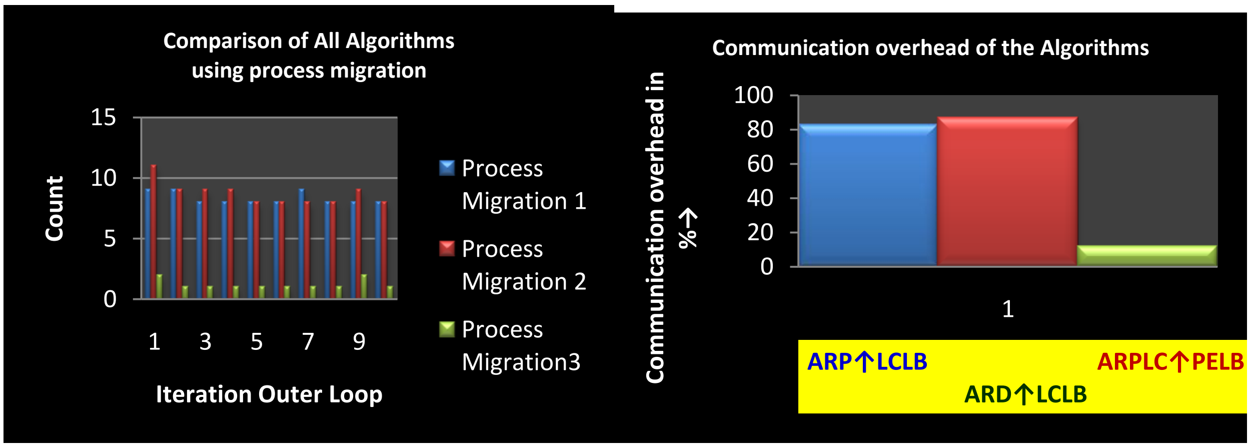

Figure 6.8 Comparison of all proposed algorithms using process migration

- In above chart, it is seen that process migration of first two algorithms is very high, but it is less in algorithm 3.
- If process migration reduces, then it again increases the overall performance of the system.

The result of the above bar chart of figure 6.52 is clearly given below by using bar chart of process migration:

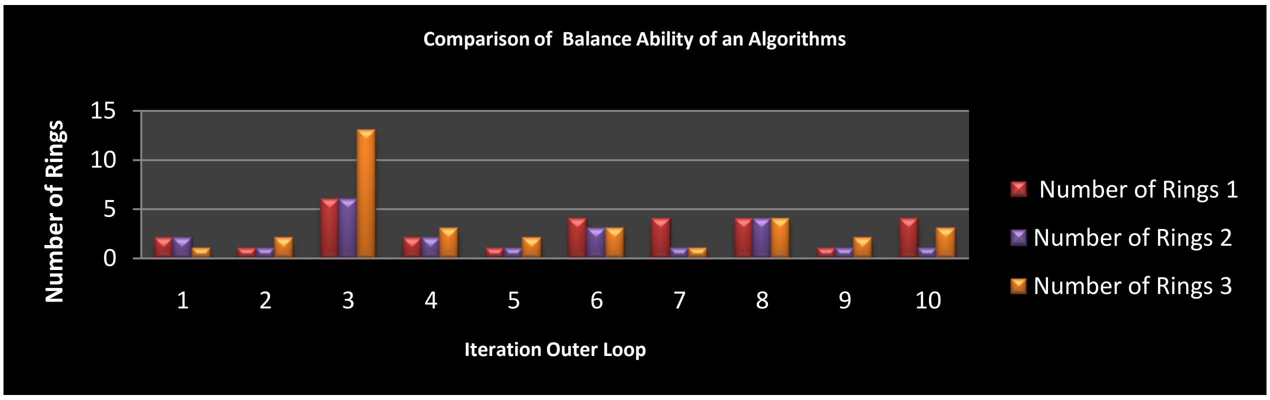

Figure 6.9 Comparison of all proposed algorithms using balance capability

- On the maximum iteration of the outer loop, the total rings of algorithm 3 are more as compare to algorithm 1 and algorithm 2, hence it is proved that system is more balanced in algorithm 3.
- It gives very good results because of decentralized logic.
- It also controls communication overhead because of the demand driven policy of the algorithm.

Now a day, the world has happened to very compact because of communication. Anybody, nevertheless computer or a human being, can communicate with each other successfully without considering the distance between them. The communication field is rapidly changing. Hence, communication between the two nodes cannot be avoided, but it can be reduced using demand driven policy.

In future it may be possible that some nodes was dedicated for communication at that time. Also, this algorithm gives very good performance. Hence it is stated that this dynamic load balancing algorithm not only increases speed of the cluster, but also provides super computing power to the cluster. The conclusion and suggestion of the research study is given in chapter.

9. Conclusion

In order to balance the load uniformly over a cluster system, our proposed algorithm has used a mix of centralized, decentralized, approach. 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 it is proved that algorithm 3 “Authority Ring with Demand Load Collection Algorithm “ gives very good results as compare to other both algorithms. It reduces process migration too. This is shown in following bar chart of process migration.

10. Future Enhancement

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 for SN to make it scalable.

11. Acknowledgment

I am sincerely express our thanks to laboratory research cell of SIBAR-MCA and SKNCOE for their full support. I am very grateful to all friends who have directly and indirectly supported to this research work

References

[1] Bernd F reisleben Dieter Hartmann Thilo Kielmann [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,Carl Kesselman, 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] Janhavi B,Sunil Surve ,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] Parimah Mohammadpour, 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] Sharada Patil, Dr Arpita Gopal,[2012], Ms Pratibha Mandave “Parallel programming through Message Passing Interface to improving performance of clusters ” – International Docteral Conference (ISSN 0974-0597) SIOM, Wadgoan Budruk in Feb 2013.