Design and Analysis of a Dynamically Reconfigurable Shared Memory Cluster

Minakshi Tripathy**
Sambalpur University, Jyotivihar, Burla, Sambalpur, Orissa,India.

Dr. C.R. Tripathy.
Department of Computer Science and Engineering V.S.S. University of Technology, Burla,Sambalpur,Orissa,India.

Abstract
In recent years, the clusters have become a viable and less expensive alternative to multiprocessor systems. This paper proposes an architecture with a load balancing and a fault tolerant model for shared memory clusters. A task clustering algorithm, a Centralized dynamic load balancing model, a load balancing algorithm and a fault tolerant model are proposed for shared memory clusters. The results establish the proposed model to provide high runtime availability and efficient load balancing.

Keywords
Cluster availability, task clustering, task allotment, load balancing, fault tolerance, checkpoint recovery.

I. INTRODUCTION

The shared memory cluster systems have become popular since they offer high computing power at low cost [1-2]. Shared memory programs are usually shorter and easier to understand then equivalent message passing programs, and large or complex data structure may easily be communicated without marshalling. Dynamic clusters are connected by a central global interconnection network. Tasks of a program are defined to prevent data cache reloading during their execution through task clustering on scheduling algorithm based on macro data flow graph representation [3-4]. Processors can be switched between clusters with data in their caches. After switching, a processor writes data from its cache to the memory allowing the data to be read on the fly by processors switching and is followed by the *"read on the fly"* called *" communication on the fly"* [5-8]. Scalability of shared memory systems can be much improved by application of cluster based system architecture. Such architecture has become quite common today [9-10]. However till date, no attempt has been made in the literature on implementation of *communication on the fly* with clustering algorithm.

Load balancing is an efficient strategy to improve the throughput or speedup execution of the set of jobs while maintaining high processor utilization [11-12]. Load balancing is broadly classified into two classes: static and dynamic. A multicomputer system with static load balancing distributes tasks across nodes using a priori known task information where the load distribution remains unchanged at run time. A multicomputer system with dynamic load balancing uses no priori task information and satisfies changing requirements by making task distribution decision during run time. In dynamic load balancing, workload is distributed among the processors at run time [13-15]. New processes are assigned to the processors based on the run time information collected from each node. If a node in the system becomes overloaded, the task that causes this overloading needs to be transferred to an under loaded node and run there. Dynamic load balancing can be further classified: centralized and decentralized dynamic load balancing. In centralized scheme all the nodes transfers their information to a cluster head for decision making. In the distributed scheme, this information is available to all the nodes. In shared memory cluster environment, the centralized scheme is more beneficial where the communication cost is less significant [16-18]. However, the method proposed in [18] suffers from certain disadvantages. The method does not report on its time complexity and the efficiency. Here, only one node acts as the central master controller called as manager. It has a global view of the load information and decides how to allot jobs to each of the nodes [19-20]. The rest of the nodes, which act as the slaves are called as workers. The workers only execute jobs assigned by the manager.

The shared memory clusters need to be fault tolerant. A fault is an anomalous physical or environmental phenomenon. Faults can be classified into transient, sticky and permanent faults [21-23]. In case of transient faults, the disk system recovers after a small finite interval e.g. link down, switch down, bus busy, parity errors, hardware or software reboot, process crash, hang, and node freeze etc. The sticky faults require human intervention for correction e.g. power failure, cable unplugging, disk hang, read and/or write fault. A permanent fault is a fault that is continuous, persistent and stable due to an irreversible change. An error is at the manifestation of a fault. It is the undesired system behaviour due to which the system is not able to deliver services. A failure is the occurrence of undesired circumstances affecting services of the system. An analytical model describes system's response to a fault. It is used to compute availability during faults with the rates of failures and repair of each component. Fault tolerance is provided through three stages including detection of faults, notification of fault followed by recovery from the fault

Manuscript received September 5, 2010
Manuscript revised September 20, 2010

[24-27]. The checkpointing is used to restore the last non-faulty state (checkpoint) of the failing task (i.e. to recover from faults). The checkpoint is saved in advance into a stable storage and is restored with event of failures of a task [28-30].

The paper is organized into four parts. In the section 2, the proposed system architecture followed by a data flow graph and a clustering algorithm for task assignment is described. In the Section 3, a centralized dynamic load balancing model is proposed. The Section 4 presents a fault tolerant model for the shared memory cluster system followed by some theoretical analysis on the cluster behaviour in case of failure of node(s). The Section 5 is devoted towards the performance analysis of the proposed system. The results presented are the performances of the proposed models are evaluated and the results are compared with the previous works [5][10][13][17][24][25]. Finally, the Section 6 provides for the conclusions.

II. Description of the Proposed Architecture

In this section, we propose a dynamically reconfigurable shared memory clusters architecture. The proposed shared memory cluster system architecture is illustrated in Fig 1.

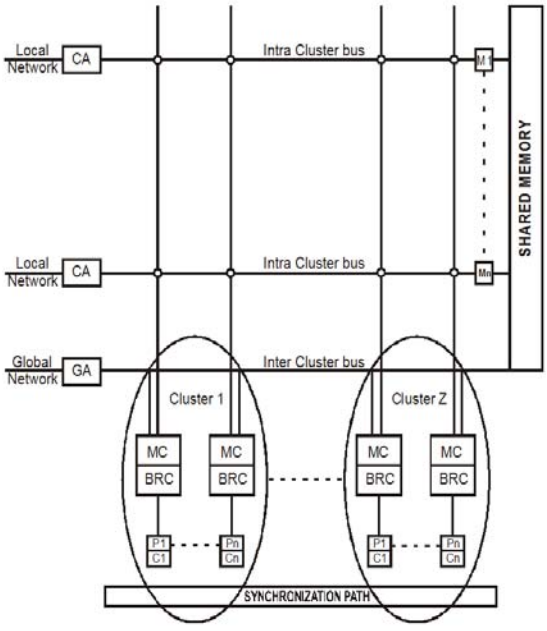

Fig. 1: Shared Memory Cluster System Architecture

The proposed system is built of a number of processors (Pi), a controller memory (CM), a set of data memory module (Mi), a set of caches and a set of buses. A memory controller arbitrates accesses to a memory module through the inter cluster bus and intra cluster buses. All the data memory modules are placed in shared address space. All the processors attached to the intra cluster bus of a data memory module constitute a processor cluster. At a time, a processor can belong to a single cluster. All processors are connected to the inter-cluster bus. Each processor data cache is connected with one module permanently during the program execution and with another module that can be changed dynamically according to the program needs. The system can contain a number of elementary modules connected by a common global network.

The permanent connection to a memory module is meant for communication with large data sets. All other processors that require to use the results need to get connected to this memory bus dynamically. While a processor writes them to the memory module through the intra cluster memory bus, other processors observe the address that appear on the bus and fetch the data they need to their data cache. Such data operation is called *read on the fly*. A read on the fly following a processor switching into a cluster is called *communication on the fly*.

Tasks of a program are defined in order to prevent data cache reloading during their execution. A processor data cache has to be filled with all necessary data before a task starts. During task execution, a processor sends computation results only to the data cache without updating the memory module. To update the memory, a processor performs a special *write* module instruction. The results that are meant for other processors are written using new addresses. Such a single assignment principle avoids data, memory and cache consistency problems. To enter a cluster, a processor performs *connect- bus* instruction. The processor is switched from using one memory module to using another. Such changes are done at the end of a computation, just before the computation result are written from the data cache to a memory module and write the result to a new memory module. In this way, a processor can be switched to a new cluster to provide module and common new data from its data cache.

Each processor is equipped with a Bus request controller (BRC). An arbiter selects the highest priority level request (first *writes* are examined and if there is no *write* then *reads* in the inter-cluster bus arbiters) and allows a processor's BRC to perform the transmission. The transmission starts only if the availability bit is set to ready. All writes and reads are acknowledged to the arbiter. If the data are unavailable, the transmission is suspended and an attempt is made for request with the same priority level. If there is no other request with the same priority or all the attempts have failed, then a negative acknowledgement is sent to the arbiter. On the *fly*, read requests are stored in the BRC in a separate bus snooping table. When BRC finds a source address on the bus equal to the source address of its move or

cache pre- fetch request stored in the table, it reads data from the bus. Then the data is sent to the data cache and the memory module according to the target address. After a read request is completed, it is removed from the table and is also removed from the request queue. The next subsection describes the task clustering on the proposed architecture.

A. Task Clustering

The initial program is first divided into sub graphs. Each such sub graph constitutes a separate parallel task. All data transfers between separate parallel tasks are executed via global communication network and creates a new larger parallel task by merging smaller tasks. The reduction in execution time is obtained by transforming global communication between separate small tasks into local communication performed inside a larger task. Such local communications may be executed *on the fly*, which further reduces their execution time. For a given task 'T', following two functions are defined.

a) F^T comp (t): It determines the execution time of task T of computation nodes on a time axis i.e. the number of potentially concurrent computations.
b) F^T comm (t):It determines the execution times of nodes in a computation graph of task T i.e. the number of concurrent communications on the fly.

For any task T, at any moment of its execution, the following function fulfills the constraints as below.

$F^T Comp(t)<= N$, $F^T comm (t)<= M$

where N is the number of processors and M is the number of shared memory modules.

1) Extended Macro-Data Flow Graph

An application program is first represented as the macro data – flow graph in which task nodes execute using data contained in processor cache. To describe activities of processors in dynamic cluster, special kinds of nodes in the program graph are introduced. Memory *read* nodes (R), memory *write* nodes (W), the intra-cluster memory bus arbiter nodes (CA), the inter-cluster global memory bus arbiter node (GA). Node R *reads* data from a memory module to the processor data cache for the subsequent task nodes. The node W *writes* data from the processor data cache to the cluster memory module. The R and W are labeled with volume of data. One *read* through the global bus and one *write* through the intra cluster bus can be done in parallel. *Writes* are done sequentially. An extended macro data flow program graph (EMDFG) transfers through the inter-cluster and intra-cluster buses is shown in Fig 2. An arbiter node is connected by bi-directional edges with many R and W nodes. It activates the node, which is ready for execution and has the highest priority. When the selected nodes are completed, it sends the token back to the arbiter. Task nodes can be mapped to the same processor. Data for task execution can be transferred through the processor cache. Then the respective write and read nodes disappear from the program graph as shown in Fig 3. A section in a program graph is a sub-graph, which is executed by a fixed subset of processors connected to the same memory module of the same cluster. After each switching of a processor to a cluster, new section is associated with section activations. The next subsection illustrates clustering mechanism with algorithms based on the macro data flow graph representation

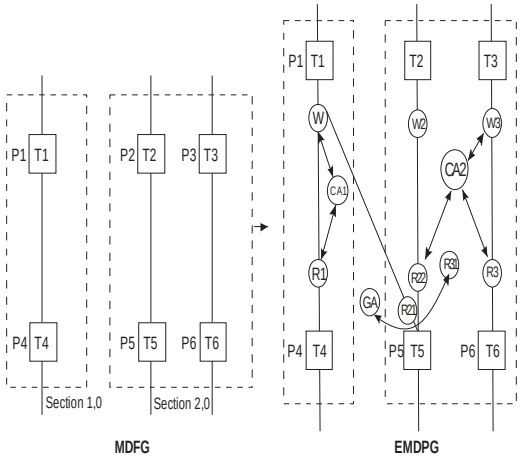

Fig. 2: a) Macro- data flow graph. b) Extended Macro data flow graph

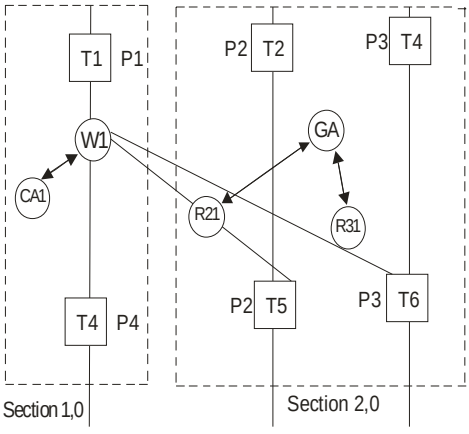

Fig. 3 : Mapping tasks onto the same processors

B. *Clustering Mechanism*

The proposed clustering method is based on the macro data flow and extended macro data flow representation of the program graph given in the Fig 2a and Fig 2b with mapping of tasks in Fig 3. It is assumed that all processors are connected to each other via a full interconnection network. If two adjacent computing nodes are mapped to

the same processor, the communication cost between them becomes zero. Otherwise, it is equal to the weight of an edge between these nodes. The proposed method has three distinct steps described below in following subsections.

a) Task cluster structuring
b) Task clustering
c) Task cluster merging

1)Task Cluster Structuring

In this subsection the proposed task cluster structuring is described followed by an algorithm. The communication subgraph (CS) of EMDFG is a subgraph containing a *read* node (R), a *write* node (W) which precedes this read in the graph and nodes of arbiters e.g CA and GA controlling transmissions. Critical path (CP) is the path going from the initial node to the end nodes whose execution time is the longest. It first selects the unexamined CS on CP. Next, a basic structure is selected which contains this CS. Finally, the selected CS is subjected to proper transformation. As a result, an equivalent program graph is obtained. An algorithm to implement the above is proposed below.

a) Algorithm (TCS)

Initialize the set S with all nodes.
Sort other read nodes from the considered CS in ascending order as per ready time (PT) and place them in a queue.
Transform the initial CS by converting all nodes from the set S to reads on the fly.
Determine execution time Te of transformed program graph
While queue Q is not empty.
  Pick the first node q from Q.
  Transform initial CS by converting all nodes from the set SU {q }
  Determine execution time t of current transformed CS.
  If t <= Te
    S=SU{q}
    Te=t
  Else
    Break the loop
  End If
End while
Set S to contain nodes to be included in a transformation
Finish

2) Task Clustering

A task clustering method supported by an algorithm is proposed below. The clustering algorithm is based on clustering technique and on observations, that converting a standard *read* operation to a *read* on the *fly* removes this *read* node from linear execution time of the graph. This *read* operation is then performed on the *fly* while the write takes place.

a) Algorithm (TS)

Set all communication sub graph (CS) as unexamined
Set parallel time (PT) as start time
While there exists an unexamined CS of CP that delay in arbiters
  Set unexamined CS with CP
    If CS's write node has one successor node
      Unify CS's write and read node cluster sequentially on
      the same processor
      Evaluate improvement of PT
      Check data cache overflows.
    Else
      Unify CS's write and read node clusters parallely on the same processor
      Or
      Unify CS's write node cluster on the same processor and read node cluster on a separate processor.
      Evaluate improvement of PT
      Check data cache overflow.
    End If
  From all clustering performed above
  Validate one with biggest PT improvement
    If PT is reduced and no data cache overflow
      Replace unified cluster by validated cluster in graph
      Set transformed PT as current PT
    End If
    If for any task T,
      $F^T comp(t) <= N$ and $F^T comm(t) <= M$
      Mark current CS as examined
      Find a new CP in the transformed graph
    Else
      Reject task T
      Mark current CS as examined
    End If
End while

3)Task Cluster Merging

This subsection proposes an algorithm for task cluster merging. The algorithm first groups the connected components of the graph in larger clusters. If the number of processors required for execution exceeds the real number of processors, loads of clusters are merged. A point wise width of the program graph is the sum of number of processors in all clusters, which co-execute in a given point of time. If a point wise width of the program graph exceeds the number of available processors, then the tasks are merged inside the processor clusters.

a) Algorithm (TCM)

Find the total number of connected components of the clustered graph CC.
If memory modules M > CC
Set CC to M i. e M= CC
Calculate sum of processors as co-execute at a particular time SP
Determine total number of real processors RP i.e..Z cluster consisting of N processor is RP= ZN
If SP<RP then
Compute PT of each CC
Merge component with smallest PT to balance PT so that CC <=Mi
M=CC
Else
Merge tasks in parallel of some cluster when no cache overflow so that SP<=M
Or
Merge tasks sequentially to balance PT of all processor in these clusters.
End If
End If

C. Theoretical Analysis

This subsection illustrates the theoretical operations on the proposed cluster. Consider a matrix multiplication operation C= AB where the order of matrices A, B and C is mxk, kxn and mxn. It follows the serial block based matrix multiplication by assuming the regular block distribution of the matrices A, B and C. Each processor accesses the appropriate blocks of the matrices A and B to multiply them together with the result stored in the locally owned part of matrix C. Our approach fetches these blocks independently, as needs without requiring any co-ordination with the processor that own the matrix blocks. The specified sequence in which the block matrix multiplications are executed is determined dynamically at run time to more efficiently schedule. For each processor P and corresponding matrix block C_{ij} is held on that processor, the following sequence is followed.

a) Build a list of tasks where a task computes each of the A_{ik} and B_{kj} products corresponding to the block matrix multiplication in

$$C_{ij}=\sum_{k=1}^{N} A_{ik} B_{kj} \qquad (1)$$

b) Reorder the task list according to the communication domains for that processes at which the A_{ik}, B_{kj} are stored.

For each task on the list,

a) Issue a non blocking involved in the next task on the list if it is not on the same node
b) Wait for the non blocking get operation bringing Aik and/or Bkj needed to execute the current task
c) Call serial matrix multiplication to compute Aik, Bkj and add the results to the Cij block.

Let us denote,

tw – data transfer time per word or element
ts – latency or startup cost
p x q – process grid in 2D fashion
P – number of processor

For our analysis, we assume a 2D matrix distribution. Each process owns a block of A, B and C matrices of size

$$\frac{m}{p}\times\frac{n}{q}, \frac{m}{p}\times\frac{k}{q} \text{ and } \frac{k}{p}\times\frac{n}{q}.$$

In a 4 x 4 grid processor P00 needs blocks of matrix A from P00, P01, P02 and P03 and blocks of matrix B from P00, P10, P20 and P30. As a further refinement, the "diagonal shift" is used to sort the task list so that the communication pattern reduces the communication contention on clusters. The node1 has processors P00, P10, P20 and P30, node2 has P01, P11, P21 and P31 etc. To compute matrix C, a processor needs the corresponding rows and columns of matrix A and B. As shown in Fig5 processor P00 needs blocks of matrix A from P00,P01,P02, and P03 and block of matrix B from P00, P10, P20 and P30. If the diagonal shift is not used processors P00, P10, P20 and P30 get a block from P01, P11, P21 and P31, in first step. Thus all the four processors are trying to share the bandwidth between node 1 and node 2. If the diagonal shift is used instead, then processors P00, P10, P20 and P30 get a block from P00 (node 1), P11 (node 2), P22 (node 3) and P33 (node 4) in first step thus reducing contention. This performs better also for more processors or nodes. The Fig 4 represents the pattern of getting block by processors in node1.

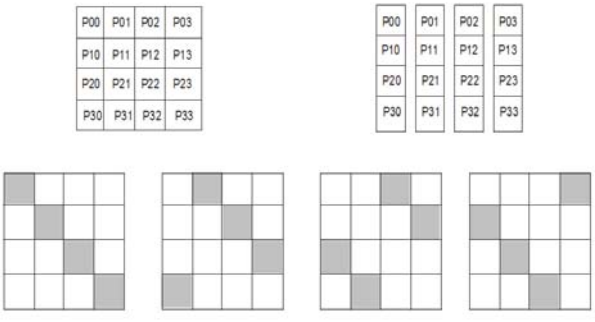

Fig. 4: Pattern of getting blocks on a 4-way cluster to reduce communication contention.

The sequential time Ts of the matrix multiplication algorithm is N^3 (say m=n=k=N).

$Ts = N^3$ (2)

The parallel time Tp is the sum of computation time (Tcomp) and the time to get the blocks of matrices A and B (Tcomm)[7].

$Tp = Tcomp + Tcomm$ (3)

Tcomm = time to get rows of matrix A block + time to get columns of B block.

$Tcomm = Trow\_comm. + Tcol\_comm$ (4)

Each process gets q blocks of matrix A and p blocks of matrix B of size $\left(\frac{m}{p}\right)\left(\frac{k}{q}\right)$ and $\left(\frac{k}{p}\right)\left(\frac{n}{q}\right)$. So,

Trow_comm = [data transfer time of message size $\frac{mk}{pq}$] + latency / startup cost

$$Trow\_comm = \left(\left(\frac{mk}{pq}\right)tw + ts\right)q \quad (5)$$

Similarly,

$$Tcol\_comm = \left(\left(\frac{nk}{pq}\right)tw + ts\right)p \quad (6)$$

Now, from equation 3, Tp = Tcomp + Tcomm

$$Tp = \frac{mnk}{p} + \left(\left(\frac{mk}{pq}\right)tw + ts\right)q + \left(\left(\frac{kn}{pq}\right)tw + ts\right)p \quad (7)$$

For simplicity let us assume m=n=k=N and p=q= $\sqrt{P}$.

Then equation 7 becomes

$$Tp = \frac{N^3}{P} + 2\frac{N^2}{\sqrt{P}}tw + 2ts\sqrt{P} \quad (8)$$

For a network with sufficient bandwidth, Ts can be neglected, as it is relatively small when compared to the total communication time. As per Amdahl's rule [6] the speedup is the ratio of sequential execution time to parallel execution time. Therefore,

$$S = \frac{Ts}{Tp} = \frac{P.N^3}{N^3 + 2N^2\sqrt{P} + 2P\sqrt{P}} \quad (9)$$

And the efficiency is defined as the ratio of speedup obtained to the number of processors used. Thus,

$$E = \frac{S}{P} = \frac{PN^3}{N^3 + 2N^2\sqrt{P} + 2P\sqrt{P}} \quad (10)$$

III. Proposed Centralized Dynamic Load Balancing Model

This section presents the details of the proposed centralized dynamic load balancing model for the shared memory cluster computing environment followed by theoretical analysis and an algorithm. The nodes are composed of various resources including processor, memory and network connectivity as shown in Fig 5. In a shared heterogeneous environment, each node differs from the other nodes with respect to their processor, memory and disk. To accomplish worker manager model, master slave paradigm is followed where a separate master program is responsible for processes (slaves) spawning data assignment and collection of results.

A. Theoretical Analysis

Next, we model the arrival process as a poisson process with service demand of the background jobs as an exponential distribution. The Fig 5 illustrates the adopted centralized dynamic load balancing using worker manager model. Here a shared memory cluster consists of a master node with a job scheduling queue with n number of arrivals. From the n number of job arrivals with arrival time (Tai) and service time (Tsi), the mean arrival time (Ta) and mean service time (Ts) can be given by

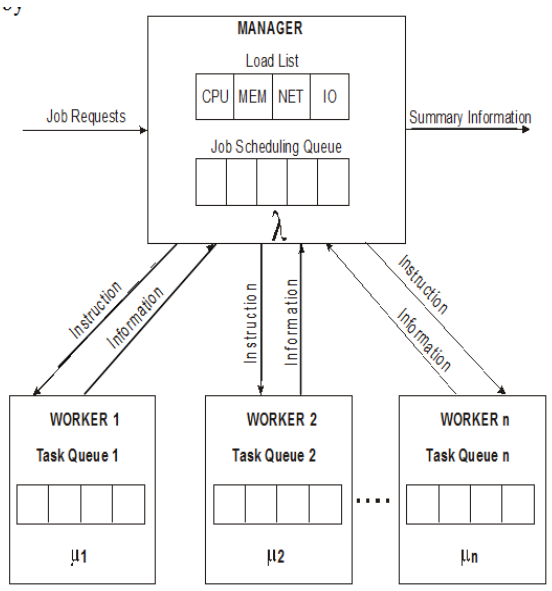

Fig.5: Centralized Dynamic load balancing worker manager model

$$Ta = \sum_{i=1}^{t} Tai \quad (11)$$

$$Ts = \sum_{i=1}^{t} Tsi \quad (12)$$

where t= Number of tasks in a job

Now the mean arrival time(Ta) and mean service time(Ts) is related with mean arrival rate(λ) and mean service time(μ) as

$$\lambda = \frac{1}{Ta} \quad (13)$$

$$\mu = \frac{1}{Ts} \quad (14)$$

In exponential distribution, traffic intensity (ρ) is

$$\rho = \frac{\lambda}{\mu} \quad (15)$$

Some principal measures of queuing system are the mean number of job requests in the queue i.e. queue length (L), mean number of requests in the queue waiting (Lw), mean time to complete service i.e. run time (W), mean time spent waiting for service to begin i.e. waiting time (Ww) can be given by

$$L = \frac{\rho}{1-\rho} \quad (16)$$

$$Lw = QL - \rho \quad (17)$$

$$W = \frac{QL}{\lambda} \quad (18)$$

$$Ww = \frac{QW}{\lambda} \quad (19)$$

Each task has a run time, which is the time period to finish the task execution and the response time(RT) is the time taken for a job to be completed after it is detects whether the task is more CPU bound, memory bound or network bound. The manager creates a temporary lookup table for the given job to submitted i.e. run time including waiting time. Hence

$$RT = W + Ww \quad (20)$$

In the proposed model, the centralized dynamic load balancing depends on some basic features such as CPU, memory and network load or any one of them considering the type of job (CPU, memory or network bound). When a new job is submitted, the manager decides on assigning tasks to workers based on CPU, memory and network load status of worker nodes. As the job continues its execution, the manager collect CPU usage of task (Wcpu), amount of demanded memory (Wmem), and amount of data transferred through network (Wnet) by the task. These parameters are stored and are applied in decision making for the next task run of the job by the manager.

$$Wcpu = \sum_{i=1}^{t} CPUload \quad (21)$$

$$Wmem = \sum_{i=1}^{t} MEMload \quad (22)$$

$$Wnet = \sum_{i=1}^{t} NETload \quad (23)$$

where, CPUload, MEMload, NETload are load or available free space of CPU, memory and network respectively. These parameters are then declared and assigned in each node for decision making by the manager. When a worker processor is ideal, the faster processor is scheduled for service before the slower processor. Now the load value (Load) of a worker node is

$$Load = Wcpu + Wmem + Wnet \quad (24)$$

The average load(Lavg) of a node can be

$$Lavg = \frac{1}{n} \sum_{j=1}^{n} Load \quad (25)$$

where t=Number of tasks executed
and n=Number of jobs completed.

Standard Deviation of load(σ) is the standard deviation of worker node's load and average load amount at every moment. It is defined as

$$\sigma = \sqrt{\frac{1}{n} \sum_{j=1}^{n} (Load - Lavg)^2} \quad (26)$$

In general, high efficiency load balancing keep up in a smaller domain along with the increase of task.

B. Description of the Proposed Algorithm

This subsection proposes an algorithm CDLBM for centralized dynamic load balancing for shared memory clusters. First, a new job j is submitted to the manager

node. Then the algorithm assumes CPU, memory and network requirements of the task i.e. the type of job whether it is CPU, memory or network bound. After finding out the highest requirement of the task, the algorithm makes an effort to balance the load. Accordingly, the tasks are allotted to a worker node for the execution of task where the expected response time is the minimum. Response time is calculated using equations(1-10). The algorithm repeatedly executes for each task of the job j. While performing the job, the lookup table status is automatically updated by the received information in every specific run. Finally the load value(Load), average load (Lavg) and standard deviation of load (σ) of each node are calculated using equations (11-16). The minimum value of the Load yields higher performance. Generally, the high efficiency load balancing makes the average load monotonically increasing in fixed percentage along with the increase of task and keeps σ in a smaller domain.

1) Algorithm (CDLBM)

For each job in the job scheduling queue of manager
Add a new job j to the manager
For each task of job j
Assume CPU, memory and network requirements
If job type (j)= Bound (CPU) then
Find a worker node where CPUload is minimum

$$CPUload(i)=\min_{i=1}^{w}(CPUload(i))$$

Calculate RT for the task j to find the worker where it
is minimum

$$\text{If } RT(i)=\min_{i=1}^{w}(RT(i)) \text{ then}$$

Allot the task to worker Wi.
End if
Else If job type (j)= Bound (MEM) then
Find a worker node where MEMload is minimum

$$MEMload(i)=\min_{i=1}^{w}(MEMload(i))$$

Calculate RT for the task j to find the worker where it
is minimum

$$\text{If } RT(i)=\min_{i=1}^{w}(RT(i)) \text{ then}$$

Allot the task to worker Wi.
End if
Else If job type (j)= Bound (NET) then
Find a worker node where NETload is minimum

$$NETload(i)=\min_{i=1}^{w}(NETload(i))$$

Calculate RT for the task j to find the worker where it
is minimum

$$\text{If } RT(i)=\min_{i=1}^{w}(RT(i)) \text{ then}$$

Allot the task to worker Wi.
End if
End If
Update the status of the lookup table
Calculate load value of each job
End For
Calculate Lavg and σ for each node
End For

The proposed CDLBM algorithm is quite efficient and has the time complexity $O(n^m)$ for m number of tasks within n number of jobs.

IV. Fault Tolerant Model

This section proposes a fault tolerant model for shared memory clusters. The goal is to achieve high performance and reliability. This includes utilization of resources and methodologies with error handling. The Fig 6 shows the architecture of our proposed fault tolerant model for shared memory clusters. We assume the model to consist of a set of N processors, interconnected by a communication channel. The processors have to access a shared memory where the code of tasks or processes and the checkpoint (last non faulty state of task) are stored. The use of shared memory checkpoints significantly reduces the task migration overhead. If a permanent error is detected on a node, the task is recovered on the other nodes from its last checkpoint stored. The status control is responsible for monitoring and updating the status of each node. The checkpointing is responsible for failure notification of each node. The status and failure information are checked before hand in order to avoid establishing communication with failed processes. The function of the control process is similar to that of a system manager.

A. Theoretical Analysis

This subsection provides the theoretical analysis for Fault tolerance. A node is considered to be in failure mode when it exhibits an abnormal behaviour in such a way that the results returned by the processor can not be used either by the remaining nodes or by the user. The time between the detection of error leading to a failure and its first occurrence is called an error detection delay.

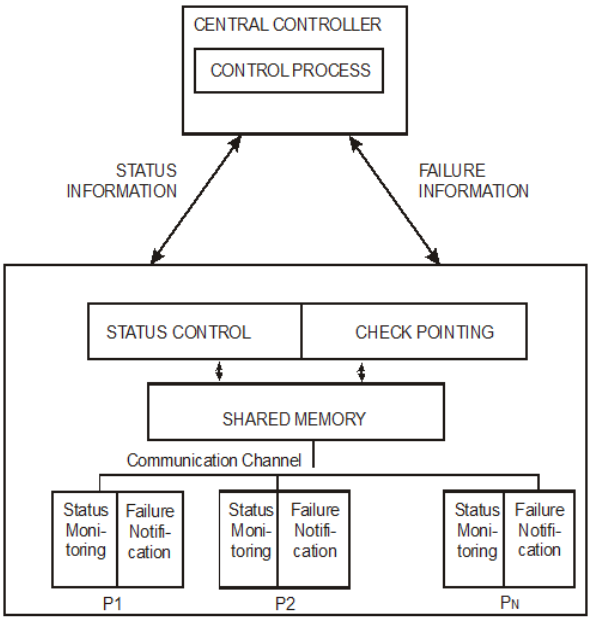

Fig.6: Fault Tolerance Model for Shared Memory Cluster

A cluster is said to be failed if i out its N nodes fail for i< N. The transaction between states is memory less i.e. it does not depend on the past states and so in order to goes back to a preceding state, a restoration process needs to be preformed. According to Poisson distribution [6-9], if X be the random variable for the number of failures of nodes, then the probability to have n failures at time interval (t) is given by

$$\Pr[X = n] = \frac{e^{-\lambda t}(\lambda t)^n}{n}, n = 0,1,2,....t > 0 \quad (27)$$

The instantaneous availability A(t) of a system is the probability that the system is operating correctly at time t, regardless of the number of times it may have failed and have been repaired in the interval(0,t).

$$A(t) = \frac{1}{T}\int_0^T A(t)\,dt \quad (28)$$

The steady state availability (SSA) is a measure of the expected fraction of time that the system is available for useful computation, and is obtained by taking the limit of A(t) when time reaches infinity.

$$SSA = \lim_{t \to \infty} A(t) \quad (29)$$

The mean time to failure(MTTF) of a system is the expected time until the occurrence of the system failure. The mean time to repair (MTTR) is a measure of the expected time for repair of a failed node. The mean time between failure(MTBF) is a measure of expected mean time between failures in a system repair and it depends on both failure and repair processes. Hence,

MTBF=MTTF+MTTR (30)

Fault arrivals are exponentially distributed and faults queue at the system so that only a single fault is in effect at any point of time t. For a repairable node with average failure rate (λ) i.e. MTBF and average repair rate(μ) i.e. MTTR, its instantaneous availability A(t) is given by

$$A(t) = \frac{\mu}{\lambda+\mu} + \frac{\lambda}{\lambda+\mu}e^{-(\lambda+\mu)t} \quad (31)$$

where $\mu = \frac{1}{MTTR}$ and $\lambda = \frac{1}{MTBF}$

The steady state availability can than be

$$SSA = \frac{\mu}{\lambda+\mu} \quad (32)$$

We assume that all the N nodes are identical and exponentially distributed with failure rate λ and repair rate μ. In our proposed architecture, using shared memory concept, all nodes are assumed both active and backup for each other. Hence, every node in the cluster of N nodes has N-1 backup nodes. So, when i number of nodes fail, the system functions with (N-i) backup nodes. The availability of the cluster system with N number of nodes is then given by

$$CA = \sum_{j=N-1}^{N} (N) A^j \bar{A}^i \quad (33)$$

where CA is the cluster availability and A, $\bar{A}$ are the availability and unavailability of a node at time t, given by

$$\bar{A}(t) = 1 - A(t) = \frac{\lambda}{\lambda+\mu} - \frac{\lambda}{\lambda+\mu}e^{-(\lambda+\mu)t} \quad (34)$$

1) Checkpointing and Recovery

This section describes the checkpointing and recovery method as a part of theoretical analysis. Once a fault is detected, a fault tolerant method needs to be invoked to handle the fault. The time needed for the detection of faults is accounted for by the error detection overhead (α). When a process is reexecuted after a fault was detected, the node restores all the initial inputs of that process. The process re-execution operation requires some time for this i.e. captured by the recovery overhead (β). In order to be restored, the initial input to a process has to be stored before

the process is executed first time. The last non faulty state or checkpoint, has to be saved in advance in the memory and will be restored if the process fails. Saving the process states including saving the initial inputs at checkpoint, takes certain amount of time known as checkpointing overhead (γ). In presence of faults checkpointing increases the task execution time. In presence of k faults, execution time (Ri) in worst case scenario of process Pi with n_i checkpoints can be obtained as below[10].

$R_i=E_i(n_i)+S_i(n_i)$ (35)

$E_i(n_i)=C_i+ n_i(\alpha_i+\gamma_i)$ (36)

$$S_i(n_i) = (\frac{C_i}{n_i} + \beta_i)k + \alpha_i(k-1) \quad (37)$$

Where $E_i(n_i)$:Execution time of process Pi with n_i checkpoints.

$S_i(n_i)$: Recovery slack of process P_i

C_i : Checkpointing cost i.e. worst case execution time of P_i.

$n_i(\alpha_i+\gamma_i)$: Overhead introduced with n_i checkpoints.

$\frac{C_i}{n_i} + \beta_i$: Time needed to recover from a single fault, when multiplied by k for recovering from k faults.

α_i : Error detection overhead

β_i : Recovery overhead

γ_i : Checkpointing overhead

Recovery slack is the ideal time on the node needed to recover the failed process segment.

B. Proposed Algorithm (SMFTC)

This subsection proposes an algorithm for checkpointing and recovery method.

For each node in the cluster
  Select process from stored list of shared memory
    Obtain recovery slack (Si), Worst case Execution Time(Ei) and Checkpointing Cost(Ci)
  For each task of a process
    For each fault of a task
      Perform Checkpointing and recovery
    End For
  Calculate Response Time of the processes.
  End For
Calculate instantaneous availability and steady state availability of the nodes.
End For
Calculate cluster availability of the system.
Finish

The proposed algorithm (SMFTC) has the time complexity of $O(n.m^k)$ for k faults with m tasks in n number of nodes.

V.PERFORMANCE EVALUATION

The matlab programming was used for the evaluation of all the theoretical analysis made in architecture, load balancing and fault tolerant sections. An instance of the program is run on a head node known as manager. It is responsible for running the proposed algorithms and gathering results from computing tasks. The manager assigns tasks to each worker by allotting data. Another instance of program is run on the worker node. It takes the tasks as multi dimensional matrix, where dimension is generated randomly for multiple numbers of jobs. It processes the data and sends the results back to the manager. To validate the effectiveness of proposed shared memory cluster architecture, a comparison is made with other architectures of previous works in SRUMMA[5] and STRASSEN[10] matrix multiplication. The proposed centralized dynamic load balancing method using worker manager model is evaluated and compared with that of the previous works in DDLB[13] and DLBM[17]. The results of proposed Shared memory fault tolerance cluster with checkpointing (SMFTC) model are compared with previous works in AMHPC [24] and RSHAC [25]. We vary the application size with several processes implemented on proposed architecture consisting of 1-1000 nodes, number of processes (1-100), number of faults(1-10) and number of checkpoints are generated randomly.

Table1 shows the performance of our architecture in milliseconds by the application of dense block matrix multiplication (DBMM). The Table2 and Table3 present speedup and efficiency improvements obtained through DBMM over the SRUMMA and STRASSEN matrix multiplication. Most of our findings show the proposed architecture provide high reduction of execution time of tasks in which speedup is an essential component. It makes the communication on the fly a promising solution to shared memory cluster architectures.

To make the effects of load balancing algorithm clear, we evaluated the response time, average load and standard deviation of load. Experimental results of executing tasks along with comparison are shown in Figure7-9. The Fig 7 compares the Response Time between DDLB and the proposed CDLBM model. The mean arrival rate, mean service rate and traffic intensity are the main factors to calculate response time. As shown in Fig 7 with the

increase in number of tasks, response time of the cluster decreases and is less then or equal to the response time of DDLB model. Therefore it improves speedup of the execution time. The Fig 8 shows the average load of proposed model with DLBM concept. Our proposed CDLBM model is found to be superior then DLBM in most of the cases. Further, it occupies fewer loads on average from CPU, memory and network avoiding system overhead. The Fig 9 shows the standard deviation of load between the proposed models with the DLBM concept. The standard deviation is the primary factor as it determines how reliable the data is. The standard deviation is more close to the average load in proposed CDLBM model as compared to that of DLBM model. This establishes the superiority of the proposed model over DLBM model in terms of system reliability and efficiency.

We consider a fault scenario with checkpoint cost for a node Ci=50ms, error detection overhead(αi)=10ms, recovery overhead(μi)=15ms and checkpointing overhead(χi)=5ms.The program is run on central node for status monitoring and fault tolerance of each node. Now to compute the response time of a process making checkpoint (ni=1,2,3) with faults (k=1,2,4,6,8,10), results are given in below Table 4-5. Here, the Mean response time of this process is found to be 5.09 ms. It is computed taking the ratio of the sum of response time of all tasks with the product of the total faults and total checkpoint. It is calculated as below.

$$Ri(mean) = \frac{\sum_{i=1}^{t}(Ri)}{\sum_{i=1}^{t}(k) * \sum_{i=1}^{t} \Sigma(n_i)} \quad (38)$$

The Table 5 shows the major faults with their noticeable MTBF and MTTR. With the MTBF and MTTR values, the average failure rate(λ), average repair rate(μ) and Cluster availability (CA) are calculated. The Fig 10 illustrates the scalability impact on cluster availability for different number of processors in the proposed SMFTC system and is also compared with the previous results in AMHPC and RSHAC. The availability decreases significantly when the number of processors increases with more number of faults affecting total runtime availability. Thus, in order to maximize the cluster availability, we need to minimize the number of faults and maximize response time. As shown in Fig 10, the proposed architecture gives high runtime availability over the previous system [24][25] with quick fault recovery and fast response time due to the proposed checkpointing and recovery method.

TABLE 1
DENSE BLOCK MATRIX MULTIPLICATION(DBMM)

N	P	Tcomp	Tcomm	Ts	Tp	S	E
64	1	262144	8194	262144	270338	0.97	.97
32	8	4096	729.73	32768	4825.74	6.8	.85
16	64	64	80	4096	144	28.4	.44
8	512	1	50.91	512	51.91	9.87	.02
4	4096	0.02	128.5	64	128.52	0.5	.0001

TABLE 2
SPEEDUP COMPARISION WITH SRUMMA

N	P	SPEEDUP		EFFICIENCY	
		SRUMMA	DBMM	SRUMMA	DBMM
1000	16	18	15.87	1.13	.9920
2000	16	25	15.94	1.56	.9960
4000	16	24	16	1.5	.9980
1000	32	20	31.64	0.63	.9888
2000	32	35	31.82	1.09	.9943
4000	32	48	31.90	1.5	.9971

TABLE3
SPEEDUP COMPARISION WITH STRASSEN

N	P	SPEEDUP		EFFICIENCY	
		Strassen	DBMM	Strassen	DBMM
512	8	17.3	7.91	2.16	.9890
1024	8	22.3	7.95	2.79	.9945
1024	16	41.3	15.88	2.58	.9922
2048	16	42.3	15.94	2.64	.9961
2048	32	43.6	31.82	1.36	.9945
4096	32	45.6	31.91	1.43	.9972

TABLE 4
RESPONSE TIME(IN MS) FOR A PROCESS

Ni	K	Ei	Si	Ri
1	1	65	65	130
1	2	65	140	205
2	4	80	190	270
2	6	80	290	370
3	8	95	323.33	418.33
3	10	95	406.67	501.67

TABLE 5 MAJOR FAULTS

Fault	MTBF	MTTR
Switch down	1 year	30 min.
Disk timeout	10 months	20 min
Link down	6 months	10 min.
Process crash	4 months	5 min
H/W reboot	2 months	2 min.

COMPARISION RESULTS

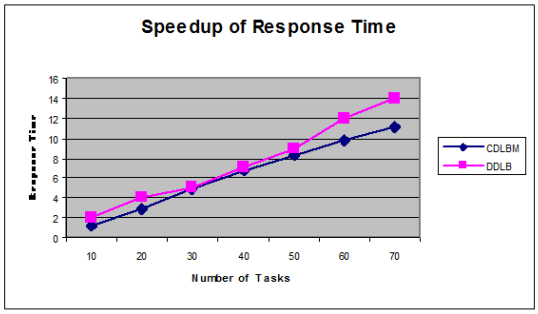

Fig. 7: Comparison of Response Time Vs Number of Tasks

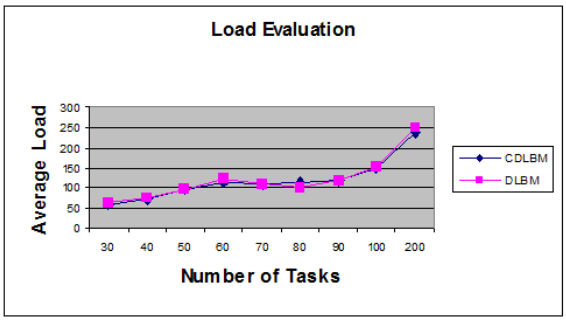

Fig. 8: Comparison of Average Load Vs Number of Tasks

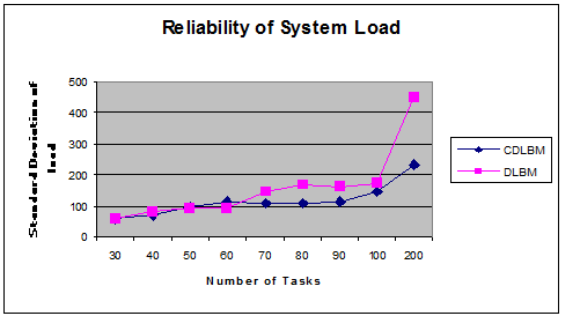

Fig. 9: Comparison of Standard Deviation of Load Vs Number of Tasks

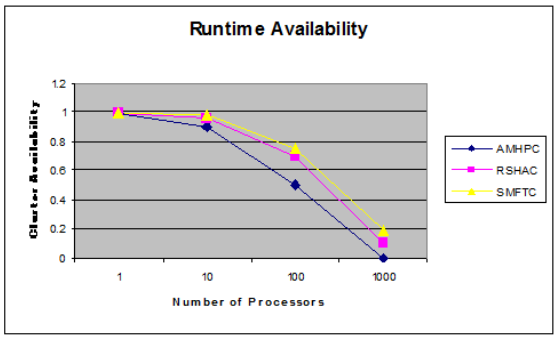

Fig. 10: Comparison of Cluster Availability Vs Number of Processor

Fig. 10: Comparison of Cluster Availability Vs Number of Processor

VI. Conclusion

The paper proposed a new architecture for dynamically reconfigurable shared memory processor clusters based on *communication on the fly*. In the proposed system, the switching between processor clusters at program run time is discussed. It is the *communication on the fly* which enables transfers of data carried in the data cache of a processor. The multiple reads on the fly are done in the cluster when the processor writes data to memory. Such a combination of processor switching and *reads on the fly* eliminates many data transactions on the buses and strongly speeds up communication in a program. It eliminates data cache reloads and thrashing. The paper also presented algorithms for scheduling program given in the form of task graphs. The algorithm uses the concept of parallel tasks. It decomposes an initial program graph to sub graphs treated as parallel tasks. The load balancing problem was discussed in detail and a new load balancing algorithm was proposed. It involves both load balancing and task allotment. Various properties of jobs such as CPU bound, memory bound or network bound were taken into consideration while deciding how to balance the load among clusters. The uniqueness of our SMFTC model is that it performs data analysis and availability modeling step by step through the proposed algorithm. The status control contains failure and repair events at various times to reflect availability information. The paper also provides availability analysis for both node wise and overall cluster system. This enables the runtime system to be aware of resource availability and ensures more accurate results with fast recovery and response from faults. Finally, the proposed architecture is compared with other clustering architecture on the basis of matrix multiplication speedup, and efficiency. The result of comparison establishes the advantages of the proposed architecture over others and illustrates the efficiency of the proposed models. It minimizes the response time of job and average load of the system, giving high speedup and avoiding system overhead with communication latency.

References

[1] Buyya R., High Performance Cluster Computing, (Vol.2, Prentece Hall PTR, New Jersey, USA 1999.)

[2] J. Protic, M. Tomasevice and V. Milutinoviec, "'A Survey of shared Memory Systems', Proc of the 28th annual Hawaii international Conference of System Sciences, Mauci, Hawai, Jan 1995, PP 74-84.

[3] T. A. Gerasoulis and T. Yang : A comparision of Clustering Heurestics for Scheduling Directed Acyclic Graphs on Multiprocessors, Journal of Parallel and Distirbuted Computing, Vol. 16, 1992, pp 276-291.

[4] M. Tudruj and L. Masko, 'An Architecture and Task Scheduling algorithms for System based on Dynamically Reconfigurable shared memory clusters', NATO Advanced Research Workshop. Advanced Environments Tools and Applications for Cluster Computing 2001, LNCS 2326, 2002, pp 197-206.

[5] M. Tudruj and L. Masko, Communication on the Fly and Program Execution Control in a System of Dynamically Configurable SMP Clusters, 11th Euromicro Conference on Parallel and Network based processing, Feb. 2003, Geneva, Italy IEEE CS press.

[6] G.M. Amdahl, Validity of the Single Procesor Approach to achieving large scale computing capabilities. In AFIPS Conference Proceedings, PP 483-485, 1967.

[7] Manoj Kumar Krishnan and Jarek Nicplocha SRUMM A: A Matrix Multiplication Algorithm suitable for clusters and scalable-shared memory systems. Proc. Of the 18th International Parallel and Distributed Processing Symposium (I PDPS 04) 2004 IEEE.

[8] B.Brutylo, M. Tudruj, L.Masko,L.Nicolas and C. Vollarie: SSOR Preconditioned Conjugate Gradient Algorithm in Dynamic SMP Clusters with Communication on the Fly, Proc. Of the 4th International Symposium on parallel and distributed Computing (ISPDC'05) 2005IEEE.

[9] M. Tudruj, and L. Masko, 'A Parallel System Architecture Based on Dynamically Configurable Shared Memory Clusters', PPAM 2001 – Parallel Processing and applied Mathematics 2001, UNCS 2328, 2002, pp 51-64.

[10] Ananth Grama, Anshul Gupta and Vipin Kumar ISO Efficiency: Measuring the scalability of parallel algorithms and architecture, IEEE Parallel and Distributed Technology, 1(3); August 1993, pp 12-21.

[11] Chau, S.-C. and Ada Wai-Chee Fu, Load balancing between computing clusters**,** Proceedings of the Fourth International Conference on " Parallel and Distributed Computing, Applications and Technologies", pp 548-551, PDCAT' Aug 2003.

[12] Tae-Hyung Kim and James M. Purtilo, Load Balancing for Parallel Loops in Workstation Clusters, Proceedings of the 1996 International Conference on Parallel Processing - Volume 3, pp 182-190, 1996.

[13] Neeraj Nehra, R.B. Patel and V.K.Bhat, 'A framework for distributed dynamic load balancing in heterogeneous cluster', Journal of Computer Science 3(1): 14-24, 2007.

[14] D.M. Tullsen, and S.J. Eggers : Effective Cache Pre-fetching on bus based Multiprocessors, ACM Trans on Computer Sytems, Vol. 13, N.I., Feb. 1995, PP – 57-88.

[15] Xiao Qin, Hong Jiang, Yifeng Zhu and David R. Swanson, 'Dynamic Load Balancing for I/O-Intensive Tasks on Heterogeneous Clusters', In Proceedinges of the 2003 International Conference on Parallel Processing Workshops. IEEE.

[16] Parimah Mohammadpour, Mohsen Sharifi and Ali Paikan, A Self-Training Algorithm for Load Balancing in Cluster Computing,Proceedings of the 2008 Fourth International Conference on Networked Computing and Advanced Information Management - Volume 01,pp104-109, 2008

[17] Huajie Zhang, On Load Balancing Model for Cluster Computers, IJCSNS International Journal of Computer Science and Network Security, VOL.8 No. 10, October 2008

[18] P. K. Chandra, B.D. Sahoo and S.K. Jena, Modeling and Analysis to Estmate the Performance of Heterogeneous Cluster, 50th Technical Review of Institution of Engineers, Orissa State Centre, pp. 113-118, 2009.

[19] Cheng-Jia Lai and Wolfgang Polak, A Collaborative Approach to Stochastic Load Balancing with Networked queues of autonomous service clusters, 2nd IEEE Collaborative Computing Conference, Atlanta, GA, USA, Nov 2006, pp 1-8.

[20] Chee Shin Yeo and Raj Kumar Buyya, Managing Risk of Inaccurate Runtime Estimates for deadline constrained job admission controls in clusters, Proceedings of the 2006 International Conference on Parallel Processing, IEEE Computer Society Washington, DC, USA, Aug. 2006, pp 451 – 458.

[21] Rajagopal Subramaniyan, Vikas Aggarwal, Adam Jacobs and Alan D. George, FEMPI: A Lightweight Fault-tolerant MPI for Embedded Cluster Systems, Proc. International Conference on Embedded Systems and Applications (ESA), Las Vegas, 3-9, 2006.

[22] Bianca Schroeder and Garth A. Gibson, A large-scale study of failures in high-performance computing systems, Proceedings of the International Conference on Dependable Systems and Networks(DSN2006), Philadelphia, PA, USA, June 25-28, 2006.

[23] Kiran Nagaraja, Xiaoyan Li, Ricardo Bianchini, Richard P. Martin and Thu D. Nguyen, Using Fault Injection and Modeling to Evaluate the Performability of Cluster- Based Services, Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4, Seattle, WA, Pages: 2 – 2, 2003

[24] Hertong Song, Chokchai "box" Leangsuksun, Raja Nassar, Narasimha Raju Gottumukkala, and Stephen Scott, Availability Modeling and Analysis on High Performance Cluster Computing Systems, The First International Conference on Availability, Reliability and Security, 2006, ARES 2006, Volume , Issue , 20-22 April 2006, Page(s): 8.

[25] Christian Kobhio, Samuel Pierre and Alejandro Quintero, Redundancy Schemes for High Availability Computer Clusters, Journal of Computer Science 2(1):33-47, 2006.

[26] Minakshi Tripathy, C.R. Tripathy and B. D. Sahoo, Job admission controls in cluster networks with queuing methods, Proceedings of the International Conference on Computing, Communication and Information Technology Applications(CCITA 2010), Jan 21-23, Page: 235-241.

[27] Minakshi Tripathy and C.R. Tripathy, Centralized Dynamic Load Balancing Model for Shared Memory Clusters,Proceedings of the International Conference on Control,Communication and Computing(ICCC 2010), Feb 18-20, Pages: 173-176.

[28] Paul Pop, Viacheslav Izosimov, Petru Eles and Zebo Peng, Design Optimization of Time-and Cost-Constrained Fault-Tolerant Embedded Systems With Checkpointing and Replication, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol-17, No.3, March 2009.

[29] Frank Liberato, Rami Melhem and Daniel Mosse, Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard Real-Time Systems, IEEE Transactions on Computers, Vol. 49, N0.9, Sept 2000.

[30] Ying Zhang and Krishnendu Chakrabarty, Fault Recovery Based on Checkpointing for Hard Real-Time Embedded Systems, Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Page: 320, 2003.

Ms. Minakshi Tripathy received the degree of B.Sc. (PCM), M.Sc. (Statistics) and MCA from Sambalpur University. She has done 'A' level course from DOEACC, New Delhi. She is currently a Ph.D. (Computer Science) student at Sambalpur University, Burla, Orissa. She has publications in two different international conferences. Her research interest includes shared memory, cluster computing, load balancing and fault tolerance.

Prof. (Dr.) C.R. Tripathy received the B.Sc. (Engg.) in Electrical Engineering from Sambalpur University and M. Tech. degree in Instrumentation Engineering from I.I.T., Kharagpur respectively. He got his Ph.D. in the field of Computer Science and Engineering from I.I.T., Kharagpur. He has more than 50 publications in different national and international Journals and Conferences. His research interest includes Dependability, Reliability and Fault–tolerance of Parallel and Distributed system. He was recipient of “Sir Thomas Ward Gold Medal” for research in Parallel Processing. He is a fellow of Institution of Engineers, India. He has been listed as leading scientist of World 2010 by International Biographical Centre, Cambridge, England, Great Britain. He was recipient of “Sir Thomas Ward Gold Med