Notice of Violation of IEEE Publication Principles

"Adaptive Load Balancing Algorithm over Heterogenous Workstations"
by Liang Guangmin
in the Proceedings of the 2008 Seventh International Conference on Grid and Cooperative Computing, October 2008, pp. 169-174

After careful and considered review of the content and authorship of this paper by a duly constituted expert committee, this paper has been found to be in violation of IEEE's Publication Principles.

This paper contains significant portions of original text from the paper cited below. The original text was copied without attribution (including appropriate references to the original author(s) and/or paper title) and without permission.

Due to the nature of this violation, reasonable effort should be made to remove all past references to this paper, and future references should be made to the following article:

"Adaptive IOCM Load Balancing Algorithm in Network of Heterogenous Workstations"
by Masani Paresh P., S. Mary Saira Bhanu, N.P. Gopalan
in the Proceedings of the 3rd International Conference on Sematics, Knowledge, and Grid October 2007, pp. 654-675

2008 Seventh International Conference on Grid and Cooperative Computing

Adaptive Load Balancing Algorithm over Heterogeneous Workstations

Liang Guangmin*
*Computer Engineering Department Shenzhen Polytechnic, Shenzhen 518055, China
Email: gmliang@oa.szpt.net

***Abstract*—Distributed computing environment consists of a collection of autonomous workstations connected by the LAN. Due to random arrival of tasks and their random CPU service time requirements, there is good possibility that several workstations are heavily loaded, while others are idle or lightly loaded. Many load sharing/balancing algorithms mainly target sharing CPU resources, and have been intensively evaluated in a homogeneous environment. The main purpose of our algorithm is to improve the overall response time of the distributed system with minimum network traffic over heterogeneous workstations. This algorithm tries to keep the load difference between any two nodes within specified thresholds, and shares CPU, Memory and IO (thus named IOCM_LB) resources with the best effort. Theoretical analysis and experimental evaluation demonstrate that compared with existing approaches, proposed algorithm can reduce the mean slow down and network traffic.**

I. Introduction

Heterogeneous Computing Environment (like Grid Environment) is a distributed group of various high performance machines connected in a suitable way to enable processing of various tasks. Diversity upon several parameters is the main feature of heterogeneous environments. A system is heterogeneous if at least one of its important characteristics is variable to the extent that it can cause heterogeneity of application execution. These characteristics can include type, speed or number of processors, memory size, etc. In this work heterogeneity is consider with variation in computing power and memory capacities of Network of workstations. A major performance objective of implementing a load balancing policy in a distributed system is to minimize execution time of each individual job, and to maximize the system throughput by effectively using the distributed resources, such as CPUs, memory modules, and I/Os. Most load sharing/balancing schemes [1], [2], [3], [4], [5] mainly consider CPU load by assuming each computer node in the system has a sufficient amount of memory space. These schemes have proved to be effective on overall performance improvement of distributed systems. However, with the rapid development of CPU chips and the increasing demand of data accesses in applications, the memory resources in a distributed system become more and more expensive relative to CPU cycles. The reasons given below are in favor to share memory and disk resources:

1. With the rapid development of RISC and VLSI technology, the speed of processors has increased dramatically in the past decade. Increasing gap in speed between processor and memory makes performance of application programs on uniprocessor, multiprocessor and distributed systems rely more and more on effective usage of their entire memory hierarchies. In addition, the memory and I/O components have a dominant portion in the total cost of a computer system.

2. The demand for data accesses in applications running on distributed systems has significantly increased accordingly with the rapid establishment of local and wide-area internet infrastructure.

3. The latency of a memory miss or a page fault is about 1000 times higher than that of a memory hit.

Figure 1 illustrates the architecture of a distributed system considered in this paper, in which each node has a combination of multiple types of resources, such as CPU, memory, network capacities and disks. In this architecture, Dynamically Scalable Closed Group of heterogeneous or homogeneous workstations has been considered. Only workstations which are in group can initiate load balancing. Every node have the AMI (all machines information, like IP address, resources attached, physical and virtual memory total size, speed of the processor, distance in the number of hops) file in which information about the other nodes available in closed group are stored.

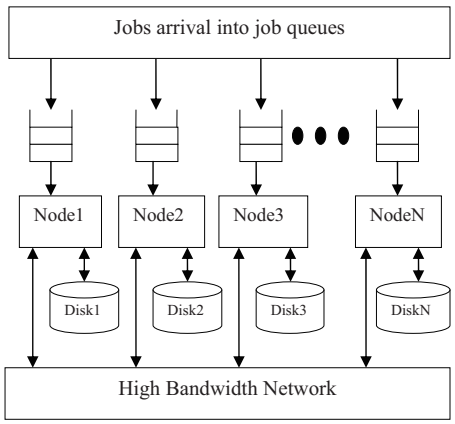

Fig. 1. Architecture of a Distributed System

Several distributed load-balancing schemes, based on the above architecture, have been presented in the literature, mainly considering CPU [1], [6], memory [7], [8], or a combination of CPU and memory [9], [10], [11]. While these load balancing policies have been by and large very effective in increasing the utilization of resources in distributed systems, they have ignored one type of resource, namely disk I/O. The

978-0-7695-3449-7/08 $25.00 © 2008 IEEE
DOI 10.1109/GCC.2008.106

impact of disk I/O on overall system performance is becoming increasingly significant as more and more data-intensive and/or I/O-intensive applications are running on distributed systems. This makes storage devices a likely performance bottleneck [12]. Therefore, by considering I/O applications proposed algorithm makes dynamic load balancing scheme more effective.

Typical examples of I/O-intensive applications include long running simulations of time-dependent phenomena that periodically generate snapshots of their state, archiving of raw and processed remote sensing data [13], [14], multimedia and web-based applications. These applications share a common feature in that their storage and computational requirements are extremely high. Therefore, the high performance of I/O-intensive applications depends heavily on the effective usage of global storage, in addition to that of CPU and memory.

The rest of the paper is organized as follows. In the Section II and Section III, related work and proposed approach in the literature is briefly reviewed. Section IV presents the IOCM_LB algorithm in detail. Section V presents the performance evaluation environment. Finally, Section VI concludes the paper by summarizing the main contributions of this paper.

II. Related work

Load balancing can be implemented through process migration or remote execution. None of traditional OS deals with process migration and because lack of commercial demand, it was mainly a research area and has not come to production use. But with the increasing deployment of distributed systems in general, and distributed operating systems in particular, the interest in process migration is again on the rise both in research and in product development. Good load balancing schemes will be on demand because high performance facilities shifts from supercomputers to Networks of Workstations (NOW) and large scale distributed systems.

Dynamic load balancing is used to improve the overall response time of the system and thus reduce the mean slowdown. Zhang et al. [9], [10], [11] focus on load sharing policies that consider both CPU and memory services among the nodes. The experimental result shown that it not only improves the performance of memory intensive jobs but also maintains the same load sharing quality of the CPU-based policies for CPU intensive jobs. In [15] Zhang also include the IO jobs, but it uses Weighted Average Load concepts and thus remote I/O accesses are prohibited, thus computation and I/O portions of a job have to be always allocated to the same node.

Many works can be found in the literature that addresses the issue of balancing the load of disk systems [19], [20], [21]. Scheuermann et al. [19] study two issues in parallel disk systems, namely striping and load balancing, and show their relationship to response time and throughput. Lee et al. [20] proposed two file assignment algorithms that minimize the variance of the service time at each disk, in addition to balancing the load across all disks. Aerts et al. [21] use randomization and data redundancy to enable effective load balancing. Since the problem of balancing the utilizations across all disks is isomorphic to multiprocessor scheduling problem [22], a greedy multiprocessor-scheduling algorithm is used.

In [12], IOCM scheme has been given which is exchanging the information periodically and thus given the solution to maintain network traffic cost under certain level. Moreover, all the studies discuss above is considering all the nodes to make the migration decision and periodic information exchange cause the worsen traffic between each nodes. Our IOCM_LB significantly enhances the overall performance of a distributed system under workload with a mixture of CPU-memory-intensive and I/O intensive jobs.

III. Proposed approach

Our IOCM_LB algorithm does not involve all the nodes but balance the load between all the nodes by cooperation of other subset of nodes during migration decision and thus reduce the tremendous traffic amount and thus leads to better performance. We used Lehmer random number generator to generate the subset of nodes. This is uniform random number generator which returns a pseudo-random number uniformly distributed 0.0 and 1.0 and based on generated number algorithm selects different node which is not already present in subset. This mechanism to generate the subset gives guarantee that the union of few subsets will form the whole system nodes and thus by this IOCM_LB able to balance the load between all the nodes available in closed group.

The IOCM_LB uses Modified Symmetric Initiated (MSI) algorithm to initiate the load balancing whenever need arise. In MSI each node maintains the receiver list (nodes which are able to accept memory, CPU or IO loads) and sender list (nodes which are able to transfer memory, CPU or IO loads) with the arrival time. These lists will be updated (may delete some entries if very old) whenever some request come for transfer or accept the load from other node.

The IOCM_LB is allows a jobs I/O operations to be conducted by a node that is different from the one in which the jobs computation is assigned, thereby permitting a job to access remote I/O. This algorithm is fully decentralized and uses variable thresholds. If the threshold is too low, the thrashing caused by excessive load balancing activities may degrade the performance significantly and if threshold is very high, effective balancing can not be carried out. Thus, whenever need arise algorithm updates the threshold values for CPU, Memory and IO. Moreover, IOCM_LB is fault tolerance one. Even though one machine crash it will not affect other machine and it will be get identify during polling. If node identifies the target node to be down it will delete the entry from the AMI as well as from its sender and receiver list. To achieve dynamism, when some node wants to join the group, it will send the all information needed to all the nodes presents in closed group. All nodes will further update the number of nodes (N).

IV. Detailed Algorithm

As stated earlier, proposed algorithm does not maintain the global snapshot for load status of nodes, but whenever

need arises it searches for the eligible node by polling subset of the nodes and thus reduces the tremendous amount of network traffic. Number of nodes in subset to be polled depends on the LB_{prob}. Thus algorithm start with the fixing up the organization requirement in percentage to find most eligible machines (LB_{prob}) and the average number of jobs (λ) available in machine. Average number of jobs is calculated by analyzing the day, evening and night workload. The probability of a machine having k jobs can be given by Poisson formula.

$$P(k) = \frac{\lambda^k e^{-\lambda}}{k!} \tag{1}$$

Let x be the average number of jobs then the probability that machine will be idle is P(0) = e^{-x} and the probability that machine is not idle (i.e. having at least one task) P(n) = 1-e^{-x} where $n \geqslant 1$. There is high probability having at least one machine eligible [23] which can accept or transfer the load when load balancing decision is made. The number of polling (P_L) needed to find the most eligible machines depends on the LB_{prob} and total number of nodes (N) available.

$$P_L = LB_{prob} * N \tag{2}$$

Let T_{lower} and T_{upper} are the threshold values to indicate a machine which is underloaded or overloaded respectively. Then probability of machine is eligible to accept the load can be given by,

$$P(eligible) = P(0) + P(T_{lower}) + 1 - P(T_{upper}) \tag{3}$$

Out of P_L machines polled, the criteria to choose the eligible machines are: 1) If machine is underloaded and ready to accept the tasks (receiver initiated) then Choose the machines with greater positive value of current load(CL) - T_{upper} or machine having current load close to TuT_{upper}. 2) If machine is overloaded and needs to transfer the tasks (sender initiated) then choose the machines which is ideal, not overloaded and smaller value of CL - T_{lower}. Proposed algorithm updates the threshold values (CPU_THRESHOLD, MEM_THRESHOLD, and IO_THRESHOLD) for the local node periodically if needed, so utilize the node effectively. Total overhead involved in migration of single task is the summation of Freezing Time (store the complete load into one file), Node Selection Time, Transfer Time, Unfreeze Time, and Unknown Overhead.

In IOCM_LB, load index considers IO, CPU and memory resources to make load balancing decision. The basic principle of this index is as follows. When a computing node in a distributed system overloaded with IO jobs, algorithm will find a node with minimum IO and minimum CPU intensive jobs and assigns computation work of IO job to node having minimum CPU intensive jobs (it may be LOCAL node) and IO task to the node having minimum IO load. If node has sufficient memory space for both running and requesting jobs, the load balancing decision is made by a CPU-based policy. When the node does not have sufficient memory space for the jobs, the system will experience a large number of page faults, resulting in long delays for each job in the node. In this case, the load sharing decision is made by a memory-based policy which attempts to migrate jobs to nodes with sufficient memory space, or even to hold the jobs in a waiting pool if necessary. The detailed pseudo code for the overloaded node of the IOCM_LB algorithm is presented in Figure 2. The code is same for under-loaded case with appropriate changes.

Initially algorithm calculates P_L by (1) and finding out the unbalanced factors by comparing current threshold values with the corresponding current load. Further steps 3 and 4 will be initiating the load balancing algorithm whenever node becomes unbalanced by memory, CPU or IO load. A node is called eligible if the load difference between source node and target node is greater than the specified LOAD_DIFF. In our simulation we set the LOAD_DIFF is equal to 2(Number of tasks in CPU queue length) for CPU, 10% for memory load and 1(Number of IO tasks) for IO jobs. Algorithm maintains load difference of any two nodes within these ranges.

$$\textbf{LBI}_{\textbf{cpu}} = \begin{cases} LocalExecution, & CT_{lower} \leqslant Q_{len} \\ & \leqslant CT_{upper} \\ Unbalanced, & otherwise \end{cases} \tag{4}$$

$$\textbf{LBI}_{\textbf{mem}} = \begin{cases} LocalExecution, & MT_{lower} \leqslant MU \\ & \leqslant MT_{upper} \\ Unbalanced, & otherwise \end{cases} \tag{5}$$

$$\textbf{MIO}_{\textbf{cost}} = \begin{cases} I_{cost}/Net\_speed, & \text{initial data has no} \\ & \text{replicated copy on remote node} \\ 0, & otherwise \end{cases} \tag{6}$$

$$\textbf{LBI}_{\textbf{new}} = \rho X LBI_{old} \tag{7}$$

Where, LBI = load balancing index, Q_{len} = CPU queue length, CPU_THRESHOLD(CT) = [1,5], Memory Usage $(MU) = \sum_{i=1}^{P} PM(i)$/Total RAM, $PM(i)$ = Memory allocated to process i, P = Total number of processes, MEM_THRESHOLD(MT) = [0.1,0.4], N_{IO}=Number of IO processes, IO_THRESHOLD(IT) = [1,3], MIO_{cost} = Cost for IO data transfer, I_{cost} = Initial data transfer cost, Net_speed = 10 Mbps, ρ=Factor by which LBI needs to be varied.

If node is underloaded for a period of time then ρ needs to be decreased and in case of overloaded its value needs to be increased. Its value for CPU, memory and IO is 1, 0.1 and 1 respectively used in IOCM_LB algorithm.

(3) Out of P_L nodes polled, source node tries to find out the target node having minimum memory requirement.

(3.1) Algorithm selects random nodes from MRL (memory receiver list), if nodes information available else selects any random node from AMI. Handshaking will be done with selected node to make sure that it is still under-loaded with memory load.

(3.2) If the selected node is still under-loaded with memory load then source node will make entry in eligible list nodes.

```
Algorithm: IOCM_LB
1. Set the LB_prob;
   P_L = LB_prob * N;
2. Find out unbalanced factors;
3. If (Unbalanced factor is memory) {
   3.1  For (i=0;i<P_L;i++) {
        If (Number of entries in MRL is not zero)
                Select one node from MRL randomly;
        Else
                Select one node from AMI randomly;
        Handshaking ();
   3.2  If (Polled node is eligible) {
                Make entry in M_eligible list;
                M_Handshaking ();
                SM_load = SM_load - TM_load ;
        } Else
                Make entry in MRL if polled node
                recommended any; // Cooperation
   3.3  If (Still Overloaded) continue; else break;
        } // end of for loop;
   3.4  For (All M_eligible nodes) {
        Decision-making-mem ();
        Confirmation message and Migration;
        }
  }
4. If (Unbalanced factor is CPU or IO) {
   4.1  If (Unbalanced factor is not CPU)
        Assign computation task of IO job to local node;
   4.2  For (i=0;i<P_L;i++) {
        If (Number of entries in IOCRL is not zero)
                Select one node from IOCRL randomly
                 based on unbalanced factor;
        Else
                Select one node from AMI randomly;
        Handshaking ();
   4.3  If ( Polled node is eligible) {
                Make entry in IOC_eligible list;
                IOC_Handshaking ();
                If ( Unbalacned factor is CPU or Computation
                        part of IO job)
                        SC_load = SC_load - TC_load ;
                Else
                        SIO_load=SIO_load - TIO_load;
        } Else
                Make entry in IOCRL if polled node
                recommended any; // Cooperation
   4.4  If (Still Overloaded) continue; else break;
        } // end of for loop;
   4.5  For (All IOC_eligible nodes) {
        Decision-making-IOC ();
        Confirmation message and Migration;
        }
```

Fig. 2. Pseudo Code of the IOCM_LB Algorithm

Now handshaking will be done for how much load target machine can accept (TM_{load}), so that source node updates the memory load (SM_{load}) by that factor. If selected machine is not eligible then it may send node list from its MRL if entry is not zero. Thus proposed algorithm is cooperative.

(3.3) If node is still overloaded with memory load then it will try to find other eligible node.

(3.4) For all eligible machines source node will make the decision how much load should transfer to which eligible node to distribute the load evenly between the nodes available in eligible list node. Here we have not considered the partition of task into subtask and so algorithm is distribute the number of tasks evenly in each node. Final transfer will be done after the confirmation from target node.

(4) If node is overloaded by IO or CPU load then it is handled here. IO jobs contain computation subtask also so whether node is overloaded with IO or CPU load, it is need to find best eligible node having CPU load minimum. It may possible that the computation task of IO jobs may assign to one node and IO task to another node. Thus a single IOCRL (IO and CPU receiver list) file is maintained.

(4.1) If CPU is not overloaded than computation part of IO job will be executed locally.

(4.2) IOCRL contains the list of the machine name, event (IO or CPU) and request arrival time. So, the selection of node and handshaking is also depends on the unbalanced factor.

(4.3) If the selected node is still underloaded with the current unbalanced factor (IO or CPU) then source node will make entry in eligible list nodes. Now handshaking will be done for how much load target machine can accept based on unbalanced factor, so that source node updates the correspondence load by that factor. If selected machine is not eligible then it may send the node list from IOCRL if its entry is not zero. So the algorithm is cooperative.

(4.4) If node is still overloaded with CPU or IO load then it will try to find other eligible node.

(4.5) Same criteria used as 3.4.

V. Performance Evaluation

Harchol-Balter and Downey [1] implemented a simulator of a homogeneous distributed system with 6 nodes where jobs executions follows round robin scheduling algorithm. The load balancing policy is based on CPU intensive jobs only. Zhang [10] modified this simulator based on CPU-memory based load balancing scheme in homogeneous environment of 6 nodes. Later on he enhanced [11] it to heterogeneous environment of 6 nodes. We further modified this simulator and developed IOCM_LB scheme to work with N nodes in heterogeneous environment. The workloads used in simulator are the 8 traces from [10], [11]. Each trace was collected from one workstation on different day time interval. We also modified this traces by including the IO jobs for N machines. Pareto distribution has been used to distribute the jobs randomly across the N nodes. Each job in workload has the following 4 items:

<arrival time, arrival node, requested memory size, duration time>

Page fault is simulated by assigning the 40% of requested memory to the each process. When the memory threshold of the jobs on node is greater or equal to the memory threshold then each process on the node will cause page fault at given page fault rate.

TABLE I
PARAMETERS USED FOR THE SIMULATED HETEROGENEOUS NETWORK OF WORKSTATIONS

CPU Speeds	100 to 800 MIPS
Memory Sizes (ms)	128 to 512 MB
Reserved space for Kernel (rk)	16 MB
User Available Space	ms - rk
Page Size	4 KB
Network Speed	10 Mbps
Page Fault Service Time	10 ms
CPU Time Slice For Each Job	110 ms
Context Switch Time	0.1 ms
Remote Execution Cost	0.3 sec
Migration Cost	0.3+D / Net_Speed
Page Fault Rate	10% Mbps
LB_{prob}	60% Mbps

The parameter values given in table 1 are mostly similar to [10], [11] for the performance comparison purpose. CPU speed is represented by Millions Instruction Per Second (MIPS). D is the total number of data bits to be transferred. IOCM algorithm uses preemption and remote execution to migrate the job based on type of job and the overhead incurred due to migration.

To calculate the CPU-Memory load, algorithm uses weight of each CPU and Memory. Standard deviation is used to calculate heterogeneity factors (H_{cpu} and H_{mem}).The CPU weight of the workstation refers to its computing capability relative to the fastest workstation in distributed system. The value of the CPU weight is less than or equal to 1. It is a relative ratio and hence it can also be represented by the CPU speed of each node measured by MIPS. If S_{cpu} represents the speed of workstation in MIPS then the CPU weight can be given as follows: $\omega_{cpu}(i) = \frac{S_{cpu}(i)}{\max_{j=1}^{P}(S_{cpu}(j))}$. Where, P is the total number of nodes available in distributed system. Similarly, the memory weight is calculated by comparing the memory sizes (S_{mem}) of the computing nodes: $\omega_{mem}(i) = \frac{S_{mem}(i)}{\max_{j=1}^{P}(S_{mem}(j))}$.

The system heterogeneity can be represented as a variance of computing powers and memory capacities. Using standard deviation and CPU weights, CPU and Memory heterogeneity can be defined as follows: $H_{cpu} = \sqrt{\frac{\sum_{i=1}^{P}(\bar{\omega}_{cpu}-\omega_{cpu}(i))^2}{P}}$, $H_{mem} = \sqrt{\frac{\sum_{i=1}^{P}(\bar{\omega}_{mem}-\omega_{mem}(i))^2}{P}}$. Where,$\bar{\omega}_{cpu}$ and $\bar{\omega}_{mem}$ are the average CPU and Memory weight respectively defines below: $\bar{\omega}_{mem} = \frac{\sum_{i=1}^{P}\omega_{mem}(i)}{P}$, $\bar{\omega}_{cpu} = \frac{\sum_{i=1}^{P}\omega_{cpu}(i)}{P}$.

A major timing measurement we have used is the mean slowdown, which is the ratio between the total wall clock time execution time of all jobs in a trace and their total CPU execution time. Major contributions to the slowdown come from the delays of page faults, waiting time for CPU service, and the overhead of migration and remote execution. Slowdown for the job i can be represent as follows:$slowdown(i) = \frac{wall\_time(i)}{CPU\_time(i)+IO\_time(i)}$, $MeanSlowdown = \frac{\sum_{i=1}^{P}(i)slowdown(i)}{P}$. Where, P is the total number of processes executed on node. The 4 platforms based on total power 3000 MIPS and the total memory capacity of 1536 MB used to evaluate the IOCM algorithm are given below:

TABLE II
FOUR PLATFORMS (P1, P2, P3 AND P4) USED TO EVALUATE PERFORMANCE OF IOCM ALGORITHM

Nodes →	N1	N2	N3	N4	N5	N6	Heterogeneity
P1 MIPS	500	500	500	500	500	500	$H_{cpu} = 0.0$
P1 RAM	256	256	256	256	256	256	$H_{mem} = 0.0$
P2 MIPS	500	500	500	500	500	500	$H_{cpu} = 0.0$
P2 RAM	512	128	128	512	128	128	$H_{mem} = 0.19$
P3 MIPS	300	700	200	800	100	600	$H_{cpu} = 0.28$
P3 RAM	256	256	256	256	256	256	$H_{mem} = 0.0$
P4 MIPS	300	700	200	800	100	600	$H_{cpu} = 0.28$
P4 RAM	512	128	128	512	128	128	$H_{mem} = 0.19$

VI. CONCLUSION AND FUTURE WORK

In this paper, we have studied adaptive load balancing scheme with objective to improve overall response time of system with minimum network traffic. Algorithm considers workload with a mixture of CPU-memory-intensive and I/O intensive jobs (thus named IOCM_LB). We have experimentally examined and compared six load balancing policies. Based on experimental analysis and results we have following conclusions:

1. Compared with the other policies, our IOCM scheme significantly enhances the overall performance of a distributed system under workload with a mixture of CPU-memory intensive and I/O intensive jobs.

2.When memory and IO jobs demand high compared with CPU jobs, the IOCM_LB outperforms all other policies.

3. When only IO jobs demand high and memory demand low, the IO_LB and IOCM_LB performs better.

4. When memory demand is high, MEM_LB, CPU_LB and IOCM_LB perform better than other policies in terms of mean slow down as well as number of page faults generated.

5. As the system heterogeneity increases, load balancing performance is affected. MEM_LB, CPU_LB, CPU_MEM performs poorly in heterogeneous environment compare to IOCM_LB.

6. IOCM_LB reduces the network traffic tremendously compare to existing periodic IOCM scheme and thus leads to better performs.

The limitations in this study:

1. The IOCM_LB assume global file system so that migrated process can continue to access the same files on all machines.

2. The algorithm is tested in simulated environment for a synthetic load.Memory and IO intensive jobs are generated by pareto distributions.

In our future work, we plan to implement same algorithm as a LINUX kernel module by considering the real applications. For further improvement in performance we also plan to add "partition of task into subtasks" feature to distribute load evenly across the nodes.

REFERENCES

[1] M. Harchol-Balter and A. B. Downey, "Exploiting process lifetime distributions for dynamic load balancing," *ACM Trans. Comput. Syst.*, vol. 15, no. 3, pp. 253–285, 1997.

[2] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "The limited performance benefits of migrating active processes for load sharing," in *SIGMETRICS*, 1988, pp. 63–72.

[3] F. Douglis and J. K. Ousterhout, "Transparent process migration: Design alternatives and the sprite implementation," *Softw., Pract. Exper.*, vol. 21, no. 8, pp. 757–785, 1991.

[4] X. Du and X. Zhang, "Coordinating parallel processes on networks of workstations," *J. Parallel Distrib. Comput.*, vol. 46, no. 2, pp. 125–135, 1997.

[5] S. Zhou, "A trace-driven simulation study of dynamic load balancing," *IEEE Trans. Software Eng.*, vol. 14, no. 9, pp. 1327–1341, 1988.

[6] C. Hui and S. Chanson, "Improved strategies for dynamic load sharing," *IEEE Concurrency*, vol. 7, p. 3, 1999.

[7] G. M. Voelker, H. A. Jamrozik, M. K. Vernon, H. M. Levy, and E. D. Lazowska, "Managing server load in global memory systems," in *SIGMETRICS*, 1997, pp. 127–138.

[8] A. Acharya and S. Setia, "Availability and utility of idle memory in workstation clusters," in *SIGMETRICS*, 1999, pp. 35–46.

[9] S. Chen, L. Xiao, and X. Zhang, "Dynamic load sharing with unknown memory demands in clusters," in *ICDCS*, 2001, pp. 109–118.

[10] L. Xiao, X. Zhang, and Y. Qu, "Effective load sharing on heterogeneous networks of workstations," in *IPDPS*, 2000, pp. 431–438.

[11] X. Zhang, Y. Qu, and L. Xiao, "Improving distributed workload performance by sharing both cpu and memory resources," in *ICDCS*, 2000, pp. 233–241.

[12] X. Qin, H. Jiang, Y. Zhu, and D. R. Swanson, "A dynamic load balancing scheme for i/o-intensive applications in distributed systems," in *ICPP Workshops*, 2003, pp. 79–.

[13] R. Ferreira, B. Moon, J. Humphries, A. Sussman, J. Saltz, R. Miller, and A. Demarzo, "The virtual microscope," in *American Medical Informatics Association, 1997 Annual Fall Symposium*, Nashville, TN, 1997, pp. 449–453. [Online]. Available: citeseer.ist.psu.edu/ferreira97virtual.html

[14] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. H. Saltz, "Titan: A high-performance remote sensing database," in *ICDE*, 1997, pp. 375–384.

[15] L. Xiao, S. Chen, and X. Zhang, "Dynamic cluster resource allocations for jobs with known and unknown memory demands," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 223–240, 2002.

[16] J. Xu and K. Hwang, "Heuristic methods for dynamic load balancing in a message-passing supercomputer," in *SC*, 1990, pp. 888–897.

[17] J. Balasubramanian, D. C. Schmidt, L. W. Dowdy, and O. Othman, "Evaluating the performance of middleware load balancing strategies," in *EDOC*, 2004, pp. 135–146.

[18] K. Benmohammed-Mahieddine, P. M. Dew, and M. Kara, "A periodic symmetrically-initiated load balancing algorithm for distributed systems," in *ICDCS*, 1994, pp. 616–623.

[19] P. Scheuermann, G. Weikum, and P. Zabback, "Data partitioning and load balancing in parallel disk systems," *VLDB J.*, vol. 7, no. 1, pp. 48–66, 1998.

[20] L.-W. Lee, P. Scheuermann, and R. Vingralek, "File assignment in parallel i/o systems with minimal variance of service time," *IEEE Trans. Computers*, vol. 49, no. 2, pp. 127–140, 2000.

[21] J. Aerts, J. H. M. Korst, and S. Egner, "Random duplicate storage strategies for load balancing in multimedia servers," *Inf. Process. Lett.*, vol. 76, no. 1-2, pp. 51–59, 2000.

[22] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[23] A. Hác and T. Johnson, "A study of dynamic load balancing in a distributed system," in *SIGCOMM*, 1986, pp. 348–356.