# Load Balancing in Clusters using optimal resource set of nodes.

**N.P.Gopalan[1] and Nagarajan.K[2]**
**Department of C.S.E., National Institute of Technology, Tiruchirappalli,**
**Tamilnadu, India – 620015**
**[1]gopalan@nitt.edu, [2]csk0303@nitt.edu**

**Abstract**

A probability model for job allocation to nodes in a cluster for better CPU utilization and reduces I/O and communication overhead is presented in this paper. For the NP-hard problem of job assignment, load balancing is done with a local and global scheduler (LS and GS). A newly arriving job need not wait at the home node for the want of resources as GS takes care of it. Using three different queues for the processes, GS assigns the best set of optimal resources at a foreign node with a minimal transfer cost to the processes. The experimentally simulated results show that the model compares well with the existing models and well suited for time and space sharing systems with minimal I/O overhead and excellent resource utilization.

**Key words:** Load balancing, causal dependency, checkpointing, process migration, process response time and remote execution.

## 1. Introduction

Clusters are becoming the primary platform for executing demanding scientific, engineering and commercial applications **[1]**. They provide a cost-effective, high-performance, time-sharing and multi-user environment for executing parallel and sequential jobs. When lengthy processes operate in a dynamic environment with continuously changing conditions, the load balancing (LB) becomes a critical issue in the clusters, where loads on nodes may not be uniform. To improve the utilization of the resources and throughput, it is essential that loads are to be balanced among the nodes. Similarly, the processes's response time and average job waiting time are to be managed efficiently. The LB may be implemented using either by Process Migration **(PM)** and /or by Remote Execution (RE) **[8]**.

During a PM, the running process is stopped and checkpointed and data are transferred to a Foreign Node ($F_N$). The PM is completed by "crashing" a process at the Home Node ($H_N$) and restarting it at the $F_N$ from the transferred checkpoint data, thus maintaining the global consistency and safe data communication as in projects such as CoCheck **[24]**. Algorithms like Charlotte **[13]**, freeze free **[21]** and MPVM **[6]** provide for safe data communication by implementing PM into the data communication protocol. In RE, when a process is transferred to another node and started, no memory is allocated for it. This causes less overhead and is significant while smaller processes are considered. This paper addresses the user-level dynamic LB based on the appropriate choices of PM and RE, which can be achieved with interdependent LS and GS. The LS contains a Ready Queue (LRQ) with a "transferring and job assignment mechanism". It runs at each node of the cluster. The job assignment is an NP-hard problem**[14]**as the job arrival time may be random with different priorities and non-linear CPU-burst rate.

To improve the process response time and throughput, the LS execute the shortest process and the independent

partial jobs of long-term process at the $F_N$. In the present work, each LS maintains the probability value to decide on the PM/RE and redirection of the job assignment to one of the best choices of $F_N$. The workload data is periodically broadcasted to the GS, which collects the information pertaining to the cluster states (Information policy) to make decisions regarding RE and PM (LB strategy). This is done in a distributed manner and hence is scalable. To make it efficient, current workload information of each node should be updated periodically and thus, the inconsistent view of the system states due to outdated information is thwarted.

## 2. Related works

Most of the clusters are used for dedicated applications, which use parallel programming languages (like PVM **[15]**, MPI **[11]**, and LAM/MPI **[22]** etc) and static job assignment (Round Robin and Random allocation) policies completely ignoring the current state of the resource utilization. On the other hand, the space sharing global resource management (Gang Scheduling and back filling with Gang scheduling) **[7, 12, 23]** partitions the clusters into disjoint sets of machines for executing parallel jobs to increase the resource utilization. This is mostly found in batch processing systems.

Dynamic and adaptive job allocation policies are used to increase the process response time in time-sharing environments. A dynamic policy (like in GLUnix **[16]**) based on the current resource availability, assigns a newly arriving job to a lightly loaded node detected using the sender or receiver initiation policies.

Mosix **[2, 3, 4]** is a Linux operating system for clusters. It implements a transparent preemptive process migration at the kernel. It uses only PM for LB. Condor, Amoeba and Mach use implicit non-preemptive LB policy **[8]**, wherein only the active processes migrate to a $F_N$. This requires either a priori information on processes's execution times or a list of migratable processes supplied by the users.

Rhodos **[9]** is a micro kernel distributed operating system with RE and PM. The LB strategy is separated from the transferring mechanism. Most of the systems allow explicit LB with the user deciding which processes to migrate and it's time instant. But with micro kernels, implementation of PM becomes involved.

## 3. The Proposed Model

In general, efficient process scheduling is involved and none of the LB-strategy is optimal **[14]**.

The following notations are used in the present work:

The Cluster has a finite set of (n+1) nodes, $N = \{N_0, N_1, \ldots, N_n\}$ in which $N_0$ denotes the home node $H_N$ and $N_i$ ($1 \le i \le n$) are $F_N$s with the known probabilities of $P_{N0}$, $P_{N1}, \ldots P_{Nn}$ for the accommodation of a newly arriving job at $N_i$. The computation of each $P_{Ni}$ is based on the availability of the (i) CPU and Memory during the process run and (ii) Number of processes waiting in queue and their properties (like waiting either for an I/O to occur or other network resources requirement, dependent computational results from other processes, etc.).

When $P_{Ni} = 0$, there are no more resources available at Ni and a newly created processes need not wait at $H_{Ni}$ and it may have to go for the Global Ready Queue (GRQ) for its execution. This is the self-refined fault tolerance at $N_i$ and it results in maximizing the performance of process execution and minimizing the job waiting time. Using either the conditions $P_{Ni} < P_{Nj}$ && Opt ($N_j$) or $P_{Nj} = 1$ (Where $N_i$ is $H_N$ and $N_j$ is a $F_N$), LS attempts to equally distribute the resources in the cluster during process run. Thus, the proposed model utilizes the unused system resources in the cluster network.

The GS has a set of priority queues $\{Q_{j,\ j=1,2,3}\}$; explicitly, these are Global Ready Queue (GRQ), Resource Acquired

Queue (RAQ) and Process Suspended Queue (PSQ). For each $j^{th}$ queue, there is a finite set of 'm' waiting processes $W_j(\{P_1^j, P_2^j, ..., P_m^j\})$. The transfer cost ($Tc_i^j$, in Seconds) of $i^{th}$ process in the $j^{th}$ queue is the set of values of $\{c_k^i\}$, where $c_k^i$ is the cost incurred while transferring the $i^{th}$ process to the $k^{th}$ $F_N$ (where 0 £ k £ n and $N_k \hat{I} H_N$). The GRQ contains a set of 'm' independent long-term processes whose executions are not started (waiting for preemptive migration). These processes require the $F_N$s with a minimal transfer cost with a maximal $P_{Ni}$ value. This is done by using the matrix, $R = [r_{ik}]$, where each $r_{ik}$ is $P_{Nk}$ for the $i^{th}$ process with minimal transfer cost when more nodes qualify. The cost of its preemptive migration = r + m/b, where r is the cost of PM from $H_N$ to $F_N$ (in Seconds), m is the memory size of migrant process (in MB) and b is the memory transfer bandwidth (in MB/ Second). Normally, the average value of r is $\frac{1}{k(k-1)} \sum_{i=1, i \neq k}^{i=n} \sum_{j=1}^{k} r_{ij}$.

**4. The Best Optimal Node with a Minimal Transfer Cost**

For each process 'i', there is a compatibility matrix $M_i = [d_{ik}^j]$, where 0 £ £ 1, that decides the PM to the $k^{th}$ $F_N$ from the $j^{th}$ queue. The value of can be computed using PACE [20], before the process execution starts. When =1, then $F_{Nk}$ is incompatible for $i^{th}$ process's migration. The choice of the best optimal resource of a node is obtained as $Opt_R(N_k) = \forall k \; Min(d_{ik}^j)$.

Initially, one of the queues from GS is chosen (based on some scheduling criteria) and the resources from possible $F_N$s are allocated to all the processes in that queue as for as possible before switching to another queue, using the function $Opt_R(N_k)$. The set of chosen $N_k$s are referred as compatible sub-set $\{x_c\}$. The set obtained by excluding $\{x_c\}$ from $F_N$s is referred as incompatible sub-set $\{y_{ic}\}$. For the next queue in GS, the sought resources are chosen from $\{y_{ic}\}$ instead of from all $F_N$s. The above process is repeated until either $\{y_{ic}\}$ becomes null or an optimal compatible subset couldn't be found from it. The entire procedures are then repeated all over again from the start (from the set of all $F_N$s) until all processes (in all the queues of GS) are assigned with optimal resources from $F_N$s.

The optimized transfer cost is computed as follows: Let the set S be possible compatible $F_N$s. Find a set $N_{Cmp} \hat{I} S$ with a minimal transfer cost. When the process $P_i^j$ ($i^{th}$ process in $j^{th}$ queue) has a compatible sub-set $x_c$ in S, call it a compatible process.

The conditional probability for the compatibility of a process

$$P_i^j \text{ is } P(x_c) = \frac{\sum_{N_k \in S}[1 - d_{ik}^j] * P_{Nk}}{\hat{P}(all\_F_N s)} \quad (1)$$

When the process $P_i^j$ is incompatible, then the sub-set $y_{ic}$ is obtained as $N_{Incmp} = \{\text{all } F_N s\} - \{N_{Cmp}\}$, the conditional probability of incompatibility of the process $P_i^j$ is

$$P(y_{ic}) = 1 - P(x_c)$$

and $$\hat{P}(all\_F_N s) = \sum_{N_k \in all\_F_N s} P_{Nk}; \quad (2)$$

From (1) and (2), the optimal transfer cost is $Opt_T(N_k) = \underset{\substack{P_i^j \in q_j, N_k \in N_{Cmp} \\ 1 \le j \le 3, 1 \le k \le l}}{Min} \{c_k^i + r * P(x_c)\}$

A process $P_j^i$ is selects a $F_N$ only if it can maximize the information per unit cost of the process. i.e.,

$$\underset{\substack{P_i^j \in q_i, N_k \in N_{Cmp} \\ 1 \le i \le 3, 1 \le i \le n}}{Max} \left\{ \frac{IPU(N_k, P_i^j)}{c_k^i + r_{ik}} \right\} \text{ Where,}$$

$$IPU(N_k, P_i^j) = \{P(x_c)\log_2 P(x_c) - P(y_{ic})\log_2 P(y_{ic})\}$$

When the processes in RAQ require the remote resources frequently, in order to reduce the I/O and communication overhead, the processes may be transferred to those $F_N$s where the resources are available to a maximum extent and for the most of the time. Let T be the total time to run

a process and a be the probability that follows an exponential distribution for it to require a remote node for a time duration $T_{opt}$ with in T. Let $T_{Store}$, $T_{Retrive}$ be the times required to store and retrieve the checkpoint image from the stable storage respectively. It can be seen that $T_{Store} = T_{Delay} + T_{Sys} + T_{Record}$. Where, $T_{Delay}$ - Delay incurred while transferring a checkpoint to a stable storage, $T_{Sys}$ - Delay in sending a system message during checkpointing and $T_{Record}$ - Delay involved while saving a checkpoint on a stable storage. Similar notations can be defined for $T_{Retrive}$. The time required to restart at $F_N$ is

$$E = (1 - \alpha T_{Opt})(T_{Store} + T_{Opt}) + \alpha T_{Opt} * (T_{Store} + T_{Opt} + T_{Retrive} + \frac{1}{2} T_{Opt})$$

and may be simplified as

$$E = T_{Store} + T_{Opt} + \alpha[T_{Retrive} T_{Opt} + \frac{1}{2} T_{Opt}^{2}].$$

total cost of PM in RAQ is $Opt_T(N_k) + E$.

The processes are always in consistent state **[10]** in GRQ and RAQ because either they await for starting or it may be waiting for an event (for I/O or remote resource) to occur. Hence, the synchronization becomes simple. On the other hand, the PSQ contains a huge number of independent processes whose executions are in progress. The processes may also migrate for reasons mentioned in **[17, 18]**. Hence, scheduling becomes a complicated issue as the nodes are in communication on the fly with the system in an inconsistent state. To schedule efficiently and elegantly synchronize, the processes in PSQ are partitioned into smaller groups based on their causal dependency [10]. If $N_n$ is the number of casually related processes and '$C_{Syn}$' is the cost required for the process synchronization, then the total cost of PM in PSQ is $Opt_T(N_k) + E + C_{Syn} * N_n$.

## 5. Experimental Results

The experiments were performed on a cluster of PCs under Linux 2.4.18. The cluster consists of 16 dual computing nodes connected by a 100 MB/s Ethernet. The computing nodes are equipped with Pentium III processor running at 1 GHz, 128 MB of main memory and 20 GB of stable storage. All program implementations use the LAM/MPI version 1.2.5. Test programs were compiled using the GNU GCC version 2.96.

In Addition to the system utilization and I/O overhead the average response time/slowdown time in time sharing systems and waiting time in space sharing system are computed and compared during PM and RE. The system utilization h is given by, $\eta = \frac{\sum_{i=1}^{m} n_i t_i}{t_m \times N}$, where, N – total number of nodes, $n_i$ is the number of nodes used by job i, $t_i$ is the execution time for the job i and $t_m$ is $\sum_{i=1}^{m} t_i$ for all m jobs.

The experiments are of two types. First, the proposed model is compared with Mosix and GLUnix of dynamic LB time-sharing systems. Second, the result of the current model is compared with Gang-Scheduling (GgS) and backfilling combined with Gang Scheduling (BFGS) in the space-sharing environments.

In a Time-Sharing Systems, a molecular dynamics (MD) program **[5]** is executed. A random generator is used to fire the jobs to the LS such that the arrivals follow a Poisson distribution. The program uses 100 iterations and 500 job arrivals with a problem size of 50,000 atoms, which occupy 5 MB memory. The MD program is modeled based on remote resources (I/O and Remote File) requirement and the minimum number of processors required by each job is uniformly distributed from 1 to 32 using the proposed GS.

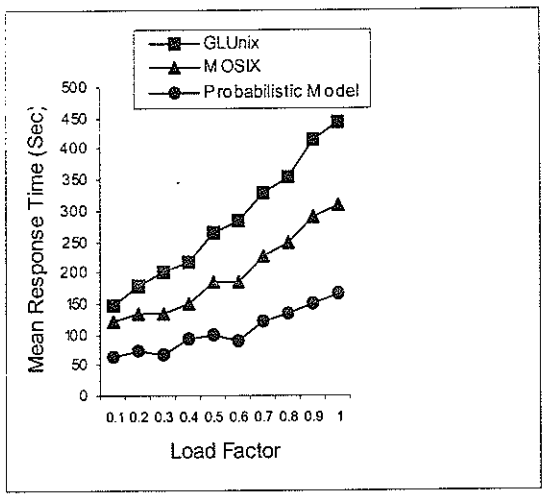


**Figure 1. Mean response time versus Load Factor using MD iterative programming**

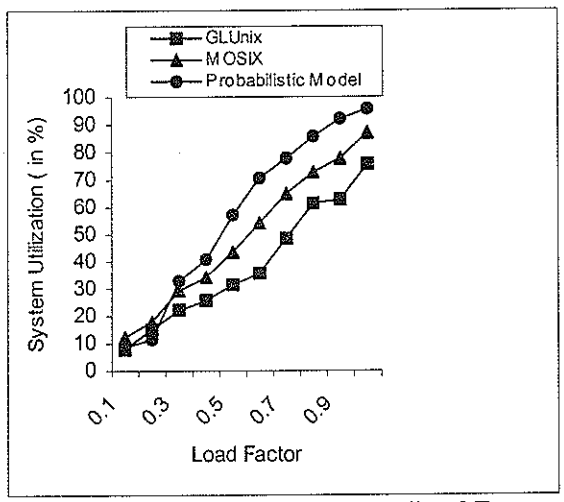


**Figure 2. System utilization versus Load Factor using MD iterative programming**

The program takes approximately 327.5 seconds to computes 100 iterations on 16 dual processors. The experiments compute the mean response time and mean system utilization. All experiments uses 500 job arrivals with 100 iterations and the results are presented in **Figure1** and **Figure 2**. The load factor (LF) is defined as, $LF = l * T_{Exc}$, where l is the mean arrival rate and $T_{Exc}$ is the execution time on 32 processors. **Figure 3** presents the mean slowdown versus I/O overhead, where each measurement is averaged over 50 different simulations. Observe that, the probabilistic model outperform GLUnix with almost 50 % improvement in I/O overhead (0.35 or more) and give slightly better results for Mosix.

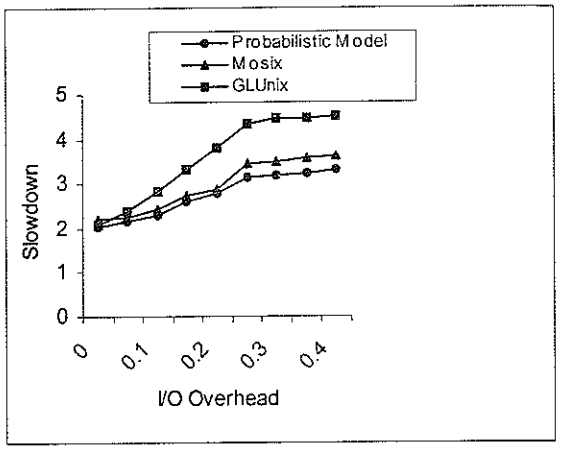


**Figure 3. Mean Slowdown versus I/O overhead in MD iterative programming**

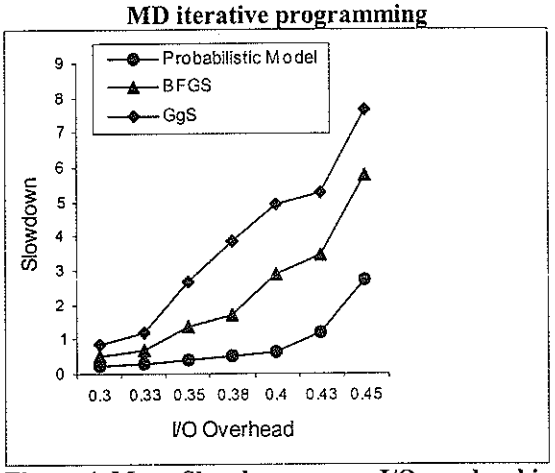


**Figure 4. Mean Slowdown versus I/O overhead in Space sharing system**

Discrete event simulation is used in space-sharing system. Each job requires a specified number of processors, runs for a certain time, and also has a user estimate for its runtime. The jobs are classified as: very short (< 30 sec), short (< 5 min), medium (< 1 hr), long (< 10 hr), and very long (> 10 hr). A single data file supplied by the user specifies its details. The batch size is set to 5,000 jobs, as recommended by MacDougall for open systems under high load **[19]**. Depending on the length of the workload, 3 to 10 batches were executed. All experiments use 32 processors with the batch size of 5,000 jobs and the results are presented in **Figure 5** and **Figure 6**. From the

results, the ability to fill the holes actually improved when the load is very heavy while using the probability models. Further, **Figure 4** presents the mean slowdown versus I/O overhead, where each measurement is averaged over 10 different batches. Observe that, the probabilistic model outer perform with 72% improvement for the I/O overhead (0.425 or higher) of GgS and give approximately 38 % better results for BFGS.

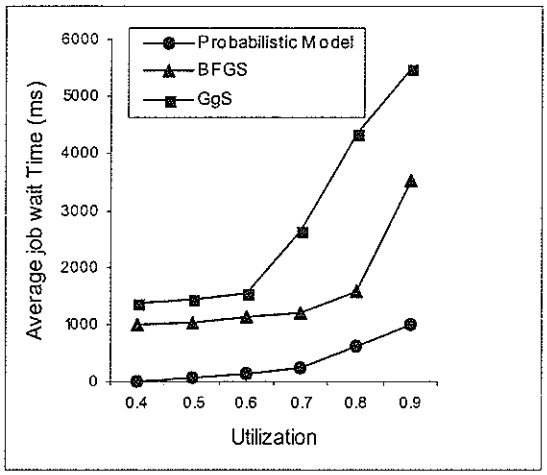


**Figure 5. Average job waiting time versus Utilization in space sharing system**

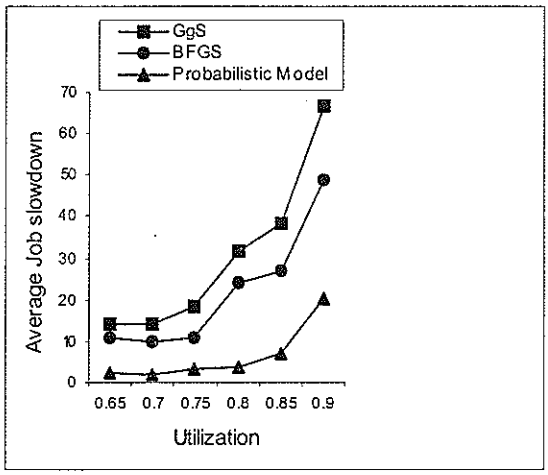


**Figure 6. Average job Slowdown versus Utilization in space sharing system**

From **Figures (1-6)**, it can be seen that the system utilization is appreciably good, possibly due to the on the fly identification of casually independent process groups in GS while scheduling. Further, the results of the present study are qualitatively the same as in all other models. Under all load factors, the I/O overhead and the average response time are less in probabilistic model implying its better suitability.

## 6. Conclusions

The experiments are conducted in space and time-sharing environments. The Local scheduler assesses for the resource availability at the home node periodically during the process run. The Global scheduler reduces the job waiting time and the average process response time by employing three different queues. The choice of best optimal resource node is determined for avoiding the frequent process migration. The processes are partitioned into smaller groups based on their causally dependency and this makes the synchronization simple while process migrate using a minimal transfer cost function. This significantly reduces the I/O and communication overhead. The mean system utilization has been observed to increase with several causally dependent processes groups.

## 7. References

[1]. T.E. Anderson, D.E. Culler, and D.A. Patterson, and the NOW team, "A Case for NOW," IEEE Micro, vol. 15, no. 1, pp. 54–64, Feb. 1995.

[2]. A. Barak, "MOSIX—Scalable Cluster Computing for UNIX," http://www.mosix.org, 2002.

[3]. A. Barak and A. Braver man, "Memory Ushering in a Scalable Computing Cluster," Microprocessors and Microsystems, vol. 22, nos. 3–4, pp. 175–182, Aug. 1998.

[4]. A. Barak and O. Ladan, "The MOSIX Multicomputer Operating System for High Performance Cluster Computing," J. Future Generation Computer Systems, vol. 13, nos. 4–5, pp. 361–372, 1998.

[5]. J. A. Board, L. V. Kale, K. Schulten, R. Skeel, and T. Schlick. "Modeling bimolecular: Larger scales, longer durations", *IEEE Computational Science and Engineering*, 1(4), 1994.

[6]. J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole, "MPVM: A Migratable Transparent Version of PVM," Computing Systems, vol. 8, no. 2, pp. 171-216, 1995.

[7]. S-H. Chiang, A. Arpaci-Dusseau, and M.K. Vernon, "The Impact of More Accurate Requested Runtimes on Production Job Scheduling Performance," Job Scheduling Strategies for Parallel Processing, pp. 103-127, Springer Verlag, 2002.

[8]. DEJAN S. Et.al. "Process Migration", ACM Computing Surveys, Vol. 32, No. 3, pp. 241–299, September 2000.

[9]. D.De.Paoli, A.Goscinski, M.Hobbs, P.Joyce: "Performance Comparison of Process Migration with Remote Process Creation Mechanisms in RHODOS", Proceedings of the IEEE 16th ICDCS '96, Hong Kong, 1996, PP 554-556.

[10]. E. Elnozahy, D. Johnson, and Y. Wang, "A survey of rollback-recovery protocols in message-passing systems", *ACM Computing Surveys (CSUR)*, Pages: 375 - 408, Volume 34, Issue 3, September 2002.

[11]. G Fagg and J.Dongarra, "Building and using a fault tolerant MPI implementation", International Journal of High performance computing applications and super computing, 2004.

[12]. D.G. Feitelson, "Packing Schemes for Gang Scheduling," Job Scheduling Strategies for Parallel Processing, pp. 89-110, Springer- Verlag, 1996.

[13]. R.A.Finkel, M.L.Scott, Y.Artsy, and H.-Y.Chang, "Experience with Charlotte: Simplicity and Function in a Distributed Operating System," IEEE Trans. Software Eng., vol. 15, no. 6, pp. 676-685, June 1989.

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

[15]. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM: Parallel Virtual Machine -A Users' Guide and Tutorial for Networked Parallel Computing", MIT Press, 1994.

[16]. D.P. Ghormley, D. Petrou, S.H. Rodrigues, A.M. Vahdat, and T.E. Anderson, "GLUnix: a Global Layer Unix for a Network of Workstations," Software-Practice & Experience, vol. 28, no. 9, pp. 929–961, 1998.

[17]. L. Gong, X.-H. Sun, and E. Watson, "Performance Modeling and Prediction of Non-Dedicated Network Computing," IEEE Trans. Computers, vol. 51, no. 9, pp. 1041-1050, Sept. 2003.

[18]. M. Harchol-Balter and A. Downey, "Exploiting Process Lifetime Distribution for Dynamic Load Balancing," ACM Trans. Computer Systems, vol. 15, 1997.

[19]. M.H. MacDougall, "Simulating Computer Systems: Techniques and Tools", MIT Press, 1987.

[20]. Nudd, G.R., Kerbyson, D.J., Papaefstatathiou, E., Et.al "PACE: a toolset for the performance prediction of parallel and distributed systems", Int. J. High Perform. Comput. Appl., 2000, 14, (3), PP 228 –251.

[21]. E. Roush, "The Freeze Free Algorithm for Process Migration," PhD thesis, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, May 1995.

[22]. J. Squyres, A. Lumsdaine, W. George, J. Hagedorn, and J. Devaney, "The Interoperable Message Passing Interface IMPI Extensions to LAM/MPI," Proc. MPI Developer's Conf., 2000.

[23]. S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "Selective Reservation Strategies for Backfill Job Scheduling," Job Scheduling Strategies for Parallel Processing, pp. 55-71, Springer- Verlag, 2002.

[24]. G. Stellner, "CoCheck: Checkpointing and process migration for MPI", In proceeding of IPPS '96. The 10th International Parallel Processing Symposium, April 15-19, Honolulu, HI, IEEE CS Press, PP 526 – 531.