http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 6 (6): 784-797, 2007
ISSN 1812-5638
© 2007 Asian Network for Scientific Information

Load Balancing with Fault Tolerance and Optimal Resource Utilization in Grid Computing

[1]Neeraj Nehra, [2]R.B. Patel and [3]V.K. Bhat
[1]School of Computer Science and Engineering,
Shri Mata Vaishno Devi University, Katra (J and K), India
[2]Department of Computer Engineering, M.M. Engineering College, Mullana (Ambala), Haryana, India
[3]School of Applied Physics and Mathematics, Shri Mata Vaishno Devi University, Katra (J and K), India

Abstract: In grid computing, load balancing with optimal resource utilization and fault tolerance are important issues. The availability of the selected resources for job execution is a primary factor that determines the computing performance. Typically, the probability of a failure is higher in the grid computing than in a traditional parallel computing and the failure of resources affects job execution fatally. Therefore, a fault tolerance service is essential in grid. Also grid services are often expected to meet some minimum levels of Quality of Service (QoS) for a desirable operation. To address this issue, we propose load balancing with optimal resource utilization and fault tolerance service that satisfies QoS requirements. A fault tolerance service deals with various types of resource failures, which include process failures, processor failures and network failures. We design and implement a fault detector and a fault manager. Approach is effective in the sense that the fault detector detects the occurrence of resource failures and the fault manager guarantees that the submitted jobs completely executed with optimal resources. The performance of job execution is improved due to job migration using Mobile Agent (MA) even if some failures occurs. This MA executes one of the check pointing algorithms and its performance is compared with check pointing algorithm-using Message Passing Interface (MPI). Also the overhead generated during job migration is compared with MA and MPI.

Key words: Fault tolerance, checkpoints, mobile agent, genetic algorithm and resource utilization

INTRODUCTION

Grid computing h as emerged as an important new field, distinguished from conventional distributed computing by its focus on large-scale resource sharing, innovative applications and high-performance orientation (Foster *et al*., 1998, 2001). The main components of a grid infrastructure are security component, resource management services, information services, data management services and load balancing (Jacob *et al*., 2003).

The real and specific problem that underlies the Grid concept is to coordinate the shared resources and to solve problems through distributed programs (Foster *et al*., 1998). The sharing that the grid computing is concerned with, is not primarily file exchange but rather direct access to computers, software data and other resources, as is required by a range of collaborative problem-solving and resource-brokering strategies. The Open Grid Services Architecture (OGSA) enables the integration of services and resources across distributed, heterogeneous, dynamic, virtual organizations-whether within a single enterprise or extended to external resource-sharing and service-provider relationships (Foster *et al*., 2001). Various grid services can be offered under the grid environment, which is defined as a web service that provides a set of well-defined interfaces and that follows specific conventions (Foster *et al*., 2002; Marovi and Javanovic, 2006). When the open grid system receives a user's request for a certain service, the Resource and Data Management Module (RDMM) starts seeking available resources, then partitions the task into multiple subtasks and assigns them to those detected resources. Many services offered by the grid need to access data from a certain source database, such as the Bio Map service using the grid system to identify the genes from open databases (Dai *et al*., 2006). A web-based grid service is defined as the grid resources which need to access visualization data from another remote server running on the grid (Marovi *et al*., 2006).

The partition of a service task into subtasks and their distribution among available resources are of great

Corresponding Author: Neeraj Nehra, School of Computer Science and Engineering, Shri Mata Vaishno Devi University, Katra (J and K), India

concern, because they significantly affect the grid service reliability, cost and profits (Boloni *et al*., 2006). A common grid service model that allowed agents representing various grid resources, which were owned by different real world enterprises (Li and Li, 2004). The grid task agents buy resources to complete tasks. Grid resource agents charge the task agents for the amount of resource capacity allocated. In the meantime, the grid task agents charge users who requested the service. (Buyya *et al*., 2005a) described the economical opportunities and realizations through grid services. They identified the challenges and requirements of economy-based grid systems and discussed various representative systems. (Li and Li, 2004; Buyya *et al*., 2005b) also introduced the optimal task/resource scheduling problems and showed the significant improvement by a good schedule strategy. Some other optimization schemes, proposed for grid are described in (Schneider, 2003; Gergel *et al*., 2005; Parashar *et al*., 2005). However, none of them consider the reliability factor when solving the optimization problems. In fact, service reliability significantly affects the profit which can be viewed as the risk cost, i.e., if certain tasks cannot be successfully finished or wrong results are offered to the users, the service providers cannot earn money, rather they may pay some penalty to compensate the users loss. Thus, when the economy is studied in the grid, the service reliability should not be ignored.

Computational grid is the most common grid (Foster *et al*., 1998), it consists of large sets of diverse, geographically distributed resources that are grouped into virtual nodes for executing specific applications. As the number in grid system components increases, the probability of a failure in the grid computing becomes higher than in a traditional parallel computing (Anh, 2000; Stelling *et al*., 1998; Vadhiyar and Dongharra, 2003; Foster *et al*., 2004). The basic component of grid is availability of resources (Li *et al*., 2007), so resource managements can encompass not only a commitment to perform a task but also commitments to level of performance or QoS (Foster *et al*., 2000).

Compute-intensive grid applications often require many hours, days, or even weeks for execution to solve a single problem and the computational grids is often hampered by their susceptibility to failures which include process failures, node crashes and network failures. Thus, appropriate mechanisms are needed for monitoring and regulating the usage of system resource to meet QoS requirements (Foster *et al*., 1998, 2004; Vraalsen *et al*., 2001; Smith, 2000). In computational grids, the fault management is a very important and difficult problem for grid application developers. Since the failure of resources affects job execution fatally, a fault tolerance service is essential in computational grids and grid applications require fault tolerance services that detect resource failures and resolve detected failures. And also grid services are often expected to meet some minimum levels of QoS for a desirable operation.

We will use the concept of Mobile Agent (MA) for optimal resource allocation in grid because Mobile Agent Technology offers a new computing paradigm in which an autonomous program can migrate under its own or host control from one node to another in a heterogeneous network. In other words, the program running at a host can suspend its execution at an arbitrary point, transfer itself to another host (or request the host to transfer it to its next destination) and resume execution from the point of suspension is called MA (Patel, 2004). MAT (Chess *et al*., 1995) provides a new solution to support load balancing. This approach consists of a number of different types of MAs in a cooperative way to fulfill the task of load balancing instead of single centralized component managing all load-balancing activities. Each type of agent implements one of the predefined policies of load balancing. Moreover, the MA paradigm supports the disruptive nature of wireless links and alleviates its associated bandwidth limitations. The migration of MA is associated with different movement costs viz, transmission time, round trip time, number of hops, etc. MA research evolved over the past few years from the creation of many monolithic Mobile Agent Systems (MASs), often with similar characteristics and built by research groups spread all over the world for optimization and better understanding of specific agent issues (Chess *et al*., 1995; Imielinsky and Badrinath, 1994). To improve the performance of MAs means to optimize their paths on the network. Furthermore, the agent uses a path through a network based upon known infrastructure characteristics. An agent optimizes its transmission between Agent hosts (AHs) (Patel, 2004; Imielinsky and Badrinath, 1994) with the help of several migration strategies described in (Al-Jaroodi *et al*., 2003; Patel and Garg, 2005).

In this study, we present an optimal load balancing with fault tolerance service using MA that detects resource failures and resolves detected failures. In order to provide a fault tolerance service that satisfies QoS requirements, we have designed a Fault Detector (FD) and a Fault Manager (FM). This article attempts to seek the optimal task partition and allocation on grid resources in order to achieve load balancing for grid service considering the effect of the reliability as well.

OVERVIEW OF PMADE

Figure 1 shows the basic block diagram of PMADE (Platform for Mobile Agent Distribution and Execution). Each node of the network has an Agent Host (AH), which is responsible for accepting and executing incoming autonomous Java agents and an Agent Submitter (AS) (Patel and Garg, 2001), which submits the MA on behalf of the user to the AH. A user, who wants to perform a task, submits the MA designed to perform that task, to the AS on the user system. The AS then tries to establish a connection with the specified AH, where the user already holds an account. If the connection is established, the AS submits the MA to it and then goes offline. The AH examines the nature of the received agent and executes it. The execution of the agent depends on its nature and state. The agent can be transferred from one AH to another whenever required. On completion of execution, the agent submits its results to the AH, which in turn stores the results until the remote AS retrieves them for the user.

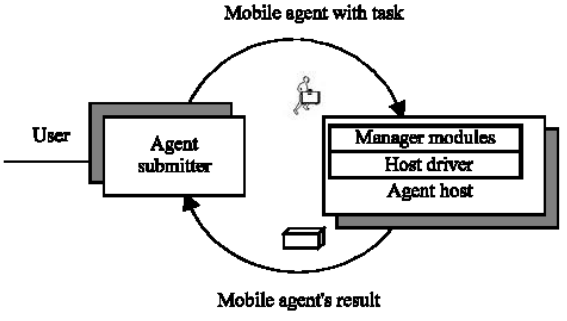

Fig. 1: Block architecture of PMADE

The AH is the key component of PMADE. It consists of the manager modules and the Host Driver. The Host Driver lies at the base of the PMADE architecture and the manager modules reside above it. It is the basic utility module responsible for driving the AH by ensuring proper co-ordination between various managers and making them work in tandem. Details of the managers and their functions are provided in (Patel and Garg, 2001). PMADE provides weak mobility to its agents and allows one-hop, two-hop and multi-hop agents (Patel and Garg, 2001). PMADE has focused on Flexibility, Persistence, Security, Collaboration and Reliability (Patel, 2004).

SYSTEM ARCHITECTURE

In Fig. 2 the architecture of the fault tolerance service has been shown. In common computational grids, resource components could be processes, processors within a computer, network interfaces, or network connections. For reasons of complexity and overhead,

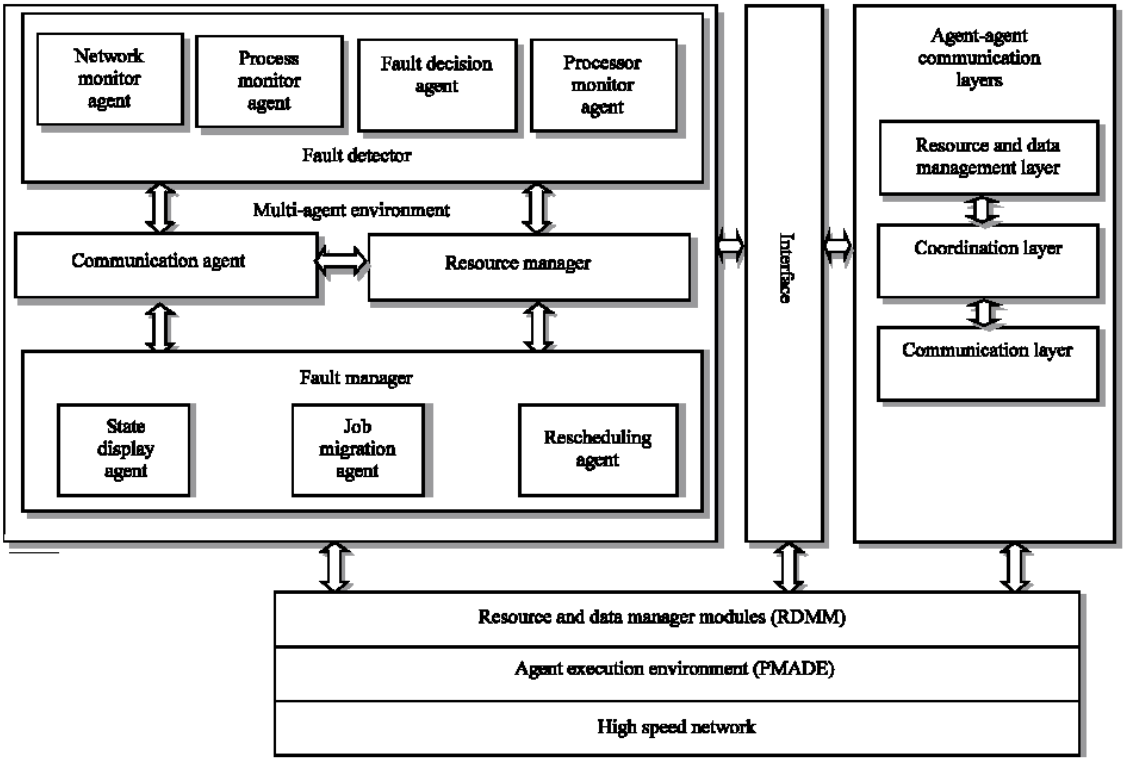

Fig. 2: Mobile agent environment

we limit the resource failure components as processes, processors and networks. If a process fails, it loses its volatile state and stops execution according to the fail stop model (Schlichting and Schneider, 1985). To provide a fault tolerance service, we propose a Fault Detector (FD) and a Fault Manager (FM). FM requests FD to monitor the state of a resource and FD periodically reports resource state information to FM. FD is responsible for monitoring the state of resources and detecting resource failures. FM is responsible for resolving detected failures. Detailed of FD and FM are as follows.

Fault detector (FD): FD decides an occurrence of a failure by analyzing the information about the state of a resource and transfers the information about the failure to FM. If FM receives the information about the failure, it tries to resolve failures. FD provide three services-monitoring, decision and communication. A monitoring service monitors resource states of processes, processors and networks. A fault detection service decides the failure occurrence for each resource. A communication service provides communication with each component. For a fault detection service, FD consists of a process monitor agent, a processor monitor agent, a network monitor agent, a fault decision agent and a communication agent. The roles of these agents are as follows:

- A process monitor agent monitors the state of a process and a starvation of a process in a job queue. The process monitor agent classifies a process state into a processing state, a stop state, a silent state and an unknown state.
- A processor monitor agent monitors the crash state of a processor (shutdown, power value) and the normal state of a processor. During the normal execution of a processor, the processor monitor agent collects the used CPU utilization and the available CPU utilization.
- A network monitor agent monitors communication bandwidth, communication latency time, network disconnection and partition between its own node and connected nodes.
- A fault decision agent decides the occurrence of a failure by analyzing state information of each resource and identifies a process failure, a processor failure, or a network failure. If a failure occurs, a fault decision agent reports the information about failure to the FM. If states information of a process is silent or unknown, the fault decision agent detects the process failure. In the case of a processor, when a fault decision agent receives the processor crash information, it detects the processor failure. If the CPU utilization is less than required quality of an application, a fault decision agent detects the processor QoS failure. When a fault decision agent receives the network disconnection or partition information, it detects the network disconnection failure. If the amount of available network bandwidth is less than the required quality of application, a fault decision agent detects the network QoS failure.
- A communication agent manages communication between agents, FM and FD. The details about the agent communication have been explained in the inter-agent communication section.

Fault manager (FM): FM provides a display service that shows each resource state to user, a job migration service for failures resolution and a communication service that provides communication with each component. For a fault management service FM consists of a state display agent, a job migration agent, a rescheduling agent and a communication agent. The roles of these agents are as follows.

- A state display agent shows the state of each resource and the type of failures occurred to the user using resource state information from FD.
- If a job migration agent receives an alert message, then it requests the rescheduling agent to decide whether job migration occurs or not. If it receives a rescheduling result for migration from the rescheduling agent, it requests to allocate new selected resources and restarts execution.
- A rescheduling agent evaluates the performance benefits that can be obtained due to job migration and decides whether job migration occurs or not. The rescheduling agent also decides a new resource allocation for jobs.

Resource manager (RM): A job requests RM for resources which execute RDMM to give resource information. The job can use the remote resources controlled by the RM. When the job uses a certain service or access remote resources in the grid, it will send its request to the RM first. Then RM will recognize those requests and match them to the service that is offered by the grid. The service will be controlled under RM to access the remote resources. When the service is completed after using remote resources, the results will be returned to the RM and then the job will get the final results from the grid service. Usually, RM is supported by multiple remote sites which may be decentralized (Krauter *et al*., 2002). These multiple remote sites can serve the job requests for services and they are backed up

to one another. If any one succeeds in providing the service, job can successfully obtain the service results, which is viewed reliable from the service perspective. These remote sites may be heterogeneous due to different locations, deployments and a variety of other factors.

Job migration with check pointing algorithm using MA: The process failure, processor failure, node crash, network failures, system performance degradation, communication delay and addition of new machines dynamically change the grid computing environment. In this computing environment, job migration is the only efficient way to guarantee that the submitted jobs are completed reliably and efficiently even though a resource failure occurs.

Fault tolerance is achieved using check pointing which is an important feature in distributed computing. It gives fault tolerance without requiring additional efforts from the programmer. A checkpoint is a snapshot of the current state of a process. It saves enough information in non-volatile stable storage such that, if the contents of the volatile storage are lost due to process failure, one can reconstruct the process state from the information saved in the non-volatile stable storage. If the process communicates with each other through message passing, rolling back a process may cause some inconsistencies. In the time since its last checkpoint, a process may have sent some messages. If it is rolled back and restarted from the point of its last checkpoint, it may create orphan messages, i.e., messages whose receive events are recorded in the states of the destination process but the send events are lost. Similarly, messages received during the rolled back period, may also cause problem. Their sending processes will have no idea that these messages are to be sent again. Such messages, whose send events are recorded in the state of the sender process but the receive events are lost, are called missing messages. A set of checkpoints, with one checkpoint for every process, is said to be consistent global check pointing state (CGCS), if it does not contain any orphan message or missing message.

Check pointing algorithms may be classified into three broad categories: (a) synchronous, (b) asynchronous and (c) quasi-synchronous (Manivannan and Singhal, 1999). In asynchronous check pointing (Meth and Tuel, 2000; Strom and Yemini, 1985) each process takes checkpoints independently. In case of a failure, after recovery, a CGCS is found among the existing checkpoints and the system restarts from there. Here, finding a CGCS can be quite tricky. The choice of checkpoints for the different processes is influenced by their mutual causal dependencies. If all the processes take checkpoints at the same time instant, the set of checkpoints would be consistent. Since globally synchronized clocks are very difficult to implement, processes may take checkpoints within an interval. In synchronous check pointing process synchronize through MPI before taking checkpoints. These synchronization messages contribute to extra overhead compared to MA approach. On the other hand, in asynchronous check pointing some of the checkpoints taken may not lie on any CGCS. Such checkpoints are called useless checkpoints. Useless checkpoints degrade system performance. Unlike asynchronous check pointing, synchronous check pointing does not generate useless checkpoints.

To overcome the above difficulties of *synchronous* and asynchronous check pointing, the MA uses quasi-synchronous check pointing algorithms (Manivannan and Singhal, 1999) in which process take checkpoints asynchronously. So there is no overhead for synchronization as in the case of MPI. Generation of useless checkpoints is reduced by forcing processes to take additional checkpoints at appropriate times. Each checkpoint (Manivannan and Singhal, 2002) is assigned a unique sequence number. The sequence number assigned to a checkpoint is the current value of a counter. The local counters maintained by the individual process are incremented periodically. The time period T_{period} is the same for all processes. Since the sequence numbers assigned to checkpoints of a process are picked from the local counters, the sequence numbers of the latest checkpoints of all the process will remain close to each other. For simplicity, we assume that each process takes checkpoints periodically with fixed time period. The gap between two checkpoints T_{period} is the same as the period for incrementing the counters. The differences in the times for checkpoints in different process will be due to the skew in their clocks. So the latest checkpoints of all process are very likely to be in CGCS.

At a point of time, a process initiate checkpoint with probability λ_C. It propagates a check pointing request to all other *n* processes, so probability that at least one process initiate check pointing is $(1-(1-\lambda_C)^n)$. Expected inter checkpoint gap = $1/1-(1-\lambda_C)^n$. Let t_c denotes average cost of taking a checkpoint. In addition to this cost there is also cost of message communication for synchronization between processes. An initiator generates n-1 checkpoint request messages and another n-1 commit messages after the acknowledgment comes back. A non initiator generates only an acknowledgment message. So the average number of messages generated per checkpoint taken is $2(n-1).\ 1/n+1.(1-1/n) = 3(n-1)/n$. Let c denotes the average cost of sending and receiving a

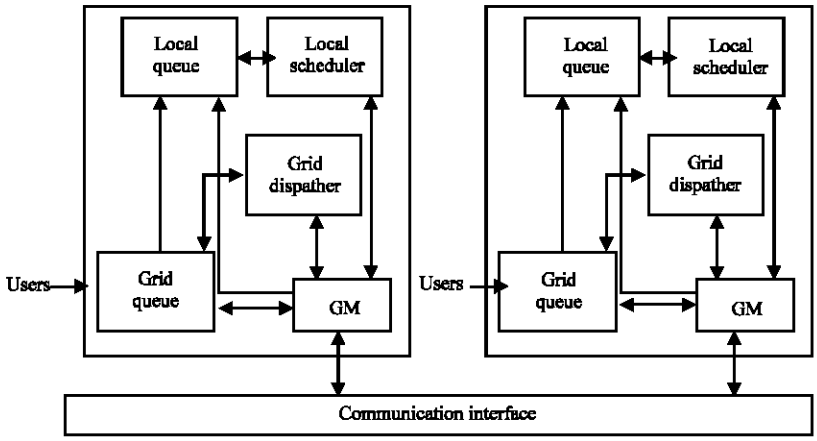

Fig. 3: Job migration scenario in typical grid computing environment

message then the average cost per checkpoint is t' = t_c+3(n-1)/n = c. So the total check pointing overhead for a process per unit time in MPI is = $(1-(1-\lambda_C)^n)t'/1+(1-(1-\lambda_C)^n)t'$.

In this algorithm, check pointing cost for a process is the sum of asynchronous check pointing cost and cost of extra checkpoint needed for establishing CGCS. Let λ' be the probability of taking a checkpoint for establishing a CGCS using MA. We can ignore the check pointing overhead cost due to forced checkpoints, because their probability is very low in this case. Hence $\lambda'<\lambda'$, which shows that overhead generated due to check pointing using MA is less compared to overhead generated using MPI in job migration.

Figure 3 shows the job migration scenario using Grid Dispatcher (GD) which calls job migration agent and rescheduling agent for job migration. Job migration agent which is MA performs the job migration using *quasi-synchronous* check pointing algorithms defined above and rescheduling agent performs rescheduling of job migration. It also decides whether job migration should occur or not. Because of job migration overhead it calculates job migration benefits that can be obtained, if resource QoS failure occurs. So in this way these two MA are involved in job migration with GD. GD is composed of a collection of autonomous local schedulers which cooperate with the dispatcher through grid middleware. A new job is first submitted to the Grid Queue (GQ), which then forwards the job's resource requirements to GD. In the distributed architecture, the GD is assumed to have a particular Local Scheduler (LS). The GD queries LS via the Grid Middleware (GM) for the Approximate Wait Time (AWT) that the job would stay in the Local Queue (LQ) before beginning execution on the local system. The LS computes the AWT based on the local scheduling policy and the LQ status. If the local resources cannot satisfy the requirements of the job, an AWT of infinity is returned. If the AWT is below a minimal threshold say ε, the job is moved from the GQ directly into the LQ without any external network communication. Otherwise, one of the below defined distributed job transfer policies is executed by agent.

Sender initiated (S-I): In the sender-initiated (S-I) policy, GD sends the resource demands of the job on each node to the RM via the communication agent. In response to this query, each node returns the AWT and Expected Run Time (ERT) of the requested job, as well as its Resource Utilization Status (RUS). Based on the collected information, the GD calculates the Turnaround Cost (TC) of each participating nodes in grid. To compute the optimal TC, first the minimum Approximate Turnaround Time (ATT) is calculated as the sum of AWT and ERT. If the minimum ATT is within a small tolerance for multiple nodes, the node with the lowest RUS is chosen to accept the job. The job is then migrated into the LQ of the machine with the minimal TC. The GM is responsible for handling the job transfer to the LQ either locally or across the communication network to a remote site. When a job enters a LQ, it will be scheduled and run based exclusively on the local policy of the LS and will no longer be controlled by the GD or migrated to another site. When the job is completed, the results are sent back to the node where it was originally submitted.

Receiver-initiated (R-I): In this policy each node in the computational grid checks its own RUS periodically at time interval λ(say). If the RUS is below a certain threshold μ, the node volunteers itself for receiving jobs by informing its neighboring node set of its low

utilization. Once a node receives this information, it checks its GQ for the first job waiting to be scheduled. If a job is indeed queued, its resource requirements are sent to the volunteer node. The underutilized node then responds with the job's ATT, as well as its own RUS. Based on this data, the node computes and compares the TC between itself and the volunteer node. If the TC of the volunteer is lower than that of this node, the job is transferred to the LQ of that node through the GM. Otherwise it continues to wait in the GQ until either its local AWT falls below ε.

Symmetrically initiated (Sy-I): As in the R-I strategy, each node periodically checks its own RUS and broadcasts a message to its partner set if it is underutilized. The difference occurs when the local AWT of a job exceeds ε but no underutilized node volunteers its services. In the R-I approach the job passively sits in the GQ while waiting for a volunteer and periodically checks its local AWT at each time interval. However, the Sy-I immediately switches to active mode and sends a request to its partners using the S-I policy. The main differences in these job migration policies therefore lie in the timing of the job transfer request initiations and the destination choice for those requests.

Central: In the centralized policy all jobs are submitted to a single GQ. GD is responsible for making global decisions and assigning each job to a particular node. GD keeps track of status of each job and maintains up to date information on all available resources, allowing it to calculate TC directly. But system having this type of policy has a single point of failure and hence suffers lack of reliability and fault tolerance.

Inter-agent communications: The framework for fault tolerance consisting of multi-agent with each agent has a specific role to play and have facility for inter agent communication as shown in Fig. 2. Each agent communicates among each other through communication agent using mobile group approach. The functions of various layers are as follows.

- **Communication and coordination layers:** Agents in the system communicate with each other or with users using mobile group approach for coordination. The request an agent receives from the communication layer should be explained and submitted to the coordination layer, which decides how the agent should act on the request according to its own knowledge. We assumed a distributed system as a collection of agents, locations and communication channels. A location represents a logical place in the distributed environment where agents execute. When a MA migrates, it moves from one location to another. Agents communicate by exchanging messages through reliable communications channels, i.e., transmitted messages are received uncorrupted and in the sequential sent order, as long as the message sender does not crash until the message is received (reliable channels can be implemented over unreliable channels by tagging transmitted messages with sequential numbers, delivering such messages according to the sequential order and asking for retransmission in case of missing messages). The failure of a given location is directly handled by FD and FM. Instead, it is only detected when the associated agents are detected faulty. An agent that never crashes is named correct. Let L denotes the set of all possible locations. Let P be the set of all possible agents. A mobile group is denoted by the set of agents $g = \{p_1, p_2 \ldots p_n\}$, $g \subset P$. On a mobile group, five operations are defined.

  - Join (g). issued by an agent, when it wants to join group g.
  - Leave (g). issued by an agent, when it wants to leave group g.
  - Move (g, l). issued when an agent wants to move from its current location to location l.
  - Send (g, m). issued by an agent when it wants to multicast a message m to the members of group g;
  - Receive (g, m). issued by an agent to receive a message m multicast from the group g.

In this way these agents communicate with each other using mobile group communication defined above for updated information about all the system resources and other valuable information (Raimundo *et al*., 2005).

- **Resource and data management layer:** This layer is responsible for management of grid resources and available data. RM manages RDMM which in turn provides all resources in the grid. It is the responsibility of RM to allocate the available resources to job in execution with the support of various agents. When any user of the grid sends a request to RM for getting a certain service then each request is assigned to execute a portion of the task. These tasks are assigned to different nodes. To reach the objectives of high service reliability and full utilization of grid resources, some tasks may have

redundancies executed by the grid. There is often a data source which manages necessary data access for a processing node. It is assumed that RDMM is to store all data for various grid services, because the major functions of RDMM focus on the partition and distribution of the jobs/programs to the resources and other data especially large datasets can be directly accessed from another database during the execution of the program. When these tasks are completed using those remote resources, the results are returned to the RM and then the users get the final results organized by the RDMM. In this case each node when executing jobs, communicates with RM for availability of resources. Because of the distributed nature of the grid, the service task allocation problem consists of dividing the task into subtasks with corresponding percentage of workload and assigning all necessary resources to execute the subtasks evoked by a service request.

OPTIMAL RESOURCE UTILIZATION USING GENETIC ALGORITHM

For finding the resource utilization we are using a genetic algorithm shown in Fig. 4. A Genetic Algorithm (GA) (Cheng *et al*., 2006) is a global search technique, which maintains a pool of potential solutions, called chromosomes (Zomaya *et al*., 2001). The GA produces new solutions through randomly combining the good features of existing solutions. This exploratory searching step is achieved by using a crossover operator, which works by randomly exchanging portions of two chromosomes (Braun *et al*., 2001). Crossover operation globally searches through the solution space. Another important local search operator is mutation, which works by randomly changing one of the genes in a chromosome. Mutation operation leads the search to get out of a local optimum. There is a selection process to remove the poor solutions. A value-based schema is used for selection. This schema probabilistically generates new population. The crossover and mutation operators are governed by their respective probabilities. The whole process is repeated a number of times, called generations or iterations. Here, each chromosome array is indexed with the resource assignment for a job. The crossover operator causes random swapping of two portions of chromosomes. Note that the crossover point is randomly chosen. A mutation randomly changes the resource assignment of a randomly selected job in an arbitrary chromosome. The crossover operator is the major facility to explore the search space to locate good solutions.

```
// Let R= set of resources {r1,r2, .,rj}, f = is the fitness threshold
value, P=Population, h = number of hypothesis to be included in
population, c = fraction of population to be replaced by crossover at
each step, m = mutation rate,  n = length of gene
GA(R, f, h c, m, n) {
// generate initial population for genetic algorithm
Initialization ( ) {
While (number of gene = h){
for (j=1; j = n; j++)
Generate new gene gi using rj in R;}
If (gi is not in P) then add this to P;}
//calculate the fitness for each gene in the population
Evaluation ( ) {
for  (I=1; I = P; I++) {
    for  (j=1; j = n; j++)
        predict execution time of rj in  gi;
  fitness of gi=Maximum execution time of rj in gi;}}
while  (Maximum( fitness of gene in P>f)){
Select ( ); Crossover ( ); Mutation( ); Evaluation( );}
// select the members of current population to add to new generation
Select ( ) {
for (I=1; I = P; I++)
if(rank of gi≤(1-c)P )add gi to new generation;}
// applying the crossover
Crossover ( ) {
for (I=1; I =cP/100; I++) {
select new gene pair from new generation;
find crossover point ck (0< k< n) at random;
for  (j=1; j≤ck; j++)
add rj in  gene pair;
for  (j=ck+1; j≤n; j++)
add rj in new gene pair; }}
// choose the members of new generation with equal probability
Mutation ( ) {
for  (I=1; I =mP/100; I++)
select gene from new generation at random;}
for  (I=1; I = P; I++)
if (rank of gene in P==1) then send this gene for resource allocation;}
```

Fig. 4: Genetic algorithm for optimal resource utilization

The algorithm consists of functions such as Initialization ( ), Evaluation ( ), Selection ( ), crossover ( ) and mutation ( ). Initialization generates an initial population P; evaluation compute fitness f of each gene in population, selection selects the number of hypothesis h to be included in current population to add to new generation. Mutation selects the members of new generation with uniform probability. The evaluation function measures the quality of a particular solution. The following parameters are adopted in this case for optimization for the number of generations:

- R = set of resources $\{r_1, r_2, ., r_j\}$.
- f = is the fitness threshold value
- P = Population
- h = number of hypothesis to be included in population
- c = fraction of population to be replaced by crossover at each step
- m = mutation rate
- n = length of gene

This GA is recursive in nature and gives the optimal solution, i.e., optimal resource allocation (Chau, 2004) for jobs in grid (Cheng *et al*., 2005). Each solution (Chau *et al*., 2003) is associated with a fitness value f, which is represented by the completion time of the schedule. Here, a chromosome is a list of ordered pairs (job ID, resource ID). The length of each chromosome can be different. Furthermore, a gene's value may be over specified, i.e., it may appear more than once in a chromosome with different values. In the scheduling context, each gene represents a (job, resource) pair. The underspecified genes do not exist in the chromosome. A randomly generated template is used for the first iteration.

Analysis of algorithm: There are some parameters in the GA such as crossover rate, mutation rate, population size, the number of iterations before termination. These parameters can affect the effectiveness of the GA. Thus, it is important to choose an appropriate group of parameters. Usually, the parameters need to be adjusted based on the outcome and performance of the experiments. Some general rules used are presented below in order to assist in quickly choosing the parameters within a few experiments. Crossover rate determines the frequency of crossover operator, which is crucial to the final optimal solution. In general, the crossover rate should be high, so a range to choose an appropriate crossover rate should be between 80 and 95% for the given problem.

Mutation is made to prevent GA from falling into local extreme, but it should not occur very often, because GA will then be converted to random search given a high mutation probability. Mutation rate determines the frequency of mutation operator, which should be very low in general and a range to choose a proper mutation rate between 0.5 and 5% for this problem.

The population size is another important parameter. It may be surprising, that very big population size usually does not improve the performance of GA (in meaning of speed of finding solution). Good population size is about 20-30, however sometimes sizes 50-120 are reported as best in some problems. The number of generations before the termination of the GA is an important parameter as well. Generally, the more generations, the better is the final solution. However, too many generations sometimes are not time-effective, especially for a large complex grid, because the solution often reaches a mature value at an early stage and then remains at the solution for many generations before another better solution appears.

Figure 5 show the decrease in execution time as the number of generates reaches the threshold. In GA the optimal solution is the fittest of the final generation, which would be many cycles of selection, mutation and crossover. As shown in Fig. 5 as the number of generates increases the execution time of job decreases. It shows that RM uses optimal resources using mutation and crossover under the available resources.

IMPLEMENTATION AND PERFORMANCE STUDY

The key measures of grid performance include the Average Response Time and the Average Wait Time. These are computed as follows (N is the total number of jobs).

$$\text{Average Response Time} = 1/N \sum_{i \in N} (\text{EndTime}_i - \text{SubmitTime}_i)$$

$$\text{Average Wait Time} = 1/N \sum_{i \in N} (\text{StartTime}_i - \text{SubmitTime}_i)$$

where SubmitTime_i, StartTime_i, EndTime_i are the times when job I is submitted to queue, when it commences execution and it is completed, respectively.

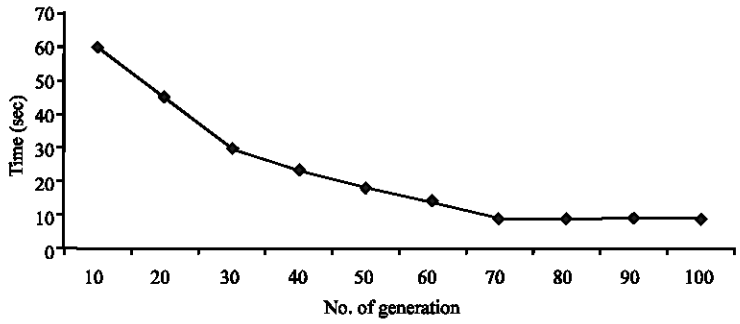

Fig. 5: The decrease in execution time as the number of generation reaches the threshold

Average Wait Time Deviation = 1/N

$$\sqrt{\sum_{i\varepsilon N}(\text{WaitTime}_i)^2} - \sqrt{(\sum_{i\varepsilon N}\text{WaitTime}_i / N)^2}$$

where $\text{WaitTime}_i = (\text{StartTime}_i - \text{SubmitTime}_i)$.

We are interested in maximizing the utilization of available resources. Thus, grid efficiency is the metric to measure overall ratio between consumed and available resources across the distributed environments. It is calculated as follows.

Grid Efficiency =

$$\frac{\sum_{i\varepsilon N}(\text{EndTime}_i - \text{StartTime}_i)\times \text{CPU}_i \times \text{CPUSpeed}_i}{(\text{EndTime}_i - \text{SubmitTime}_i)\times \sum \text{CPUm} \times \text{CPUSpeed}_m}$$

where m ε server and $(\text{EndTime}_i\text{-SubmitTime}_i)$ is the duration for entire experiment, CPU_i and CPUSpeed_i are the number of processors used by job I and their clock speed and CPU_m and CPUSpeed_m are the number of processors in node m and their clock speed.

Finally the fraction of jobs transferred for each scheduling approach is given by Fraction of jobs transferred = Number of jobs Transferred/Total number of jobs.

Figure 6 compares the job migration policies S-I, R-I, Sy-I and Central. S-I policy improves the average response time (seconds) significantly under varying system load. Under light load all policies are almost identical in terms of response time, but the key difference is under heavy load where S-I policy outperforms in comparisons of R-I and central.

Figure 7 compares the job migration policies S-I, R-I, Sy-I and Central. S-I policy improves the average wait time (seconds) compared to R-I under varying system load. Also as the number of nodes increases in grid MA becomes more effective even for lightly loaded case.

Figure 8 compares fraction of job migration (%) under varying load. We find that R-I performance is lower than that of S-I. This is because R-I approach is more passive than S-I, waiting for node to take initiative and thus migrating only few number of jobs. Sy-I is more flexible than R-I, having the option of passively wait for node to advertise their availability or to actively migrate jobs if no volunteers appears. The Sy-I have a good balance achieving better performance than R-I but transfer lesser jobs than S-I. The central approach has almost same performance as that of S-I in heavy load, because centralized approach has only single GQ but S-I has multiple. Also in S-I a job is considered for migration only if its AWT is larger than a threshold but in central approach all jobs are assigned to node based on TC. Also

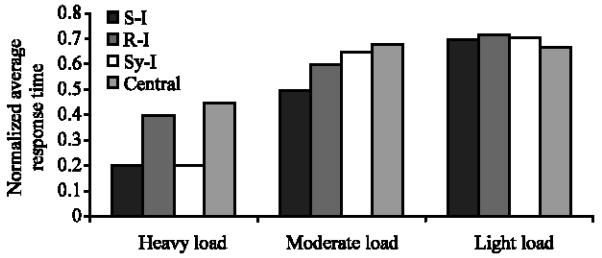

Fig. 6: Normalized average response time under varying load

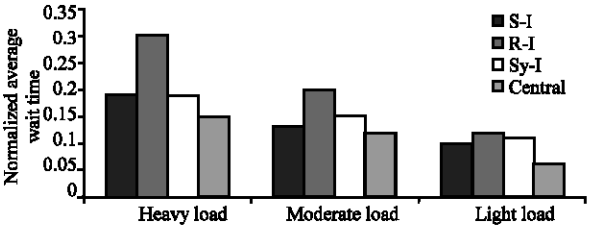

Fig. 7: Normalized averages wait time under varying load

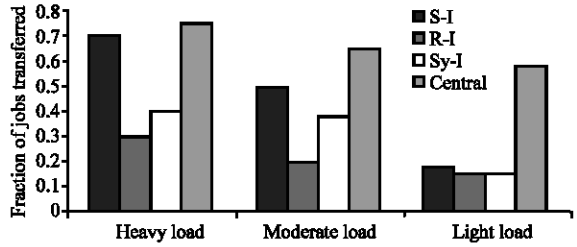

Fig. 8: Fraction of jobs transferred under varying load

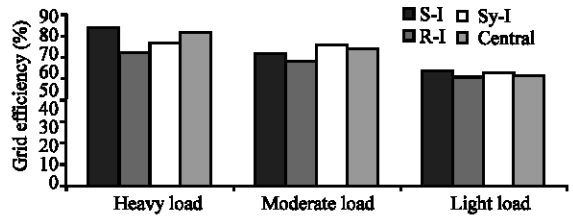

Fig. 9: Grid efficiency under varying load

central approach is too limited in terms of fault tolerance and scalability. So S-I is more conservative in moving the jobs. All these policies are executed by MA using quasi-synchronous check pointing algorithm.

As shown in Fig. 9 there is almost no change for lightly loaded case. But for heavily loaded case grid efficiency is increased to 82% using S-I compared to 70 and 75% using R-I and Sy-I policy, respectively. Even though there is a little change in grid efficiency for light

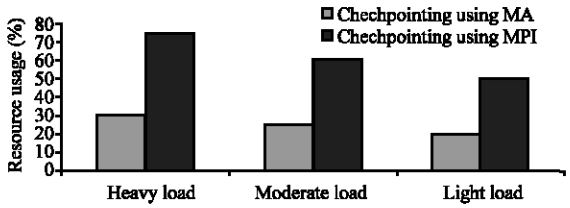

Fig. 10: The resource utilization under varying system load

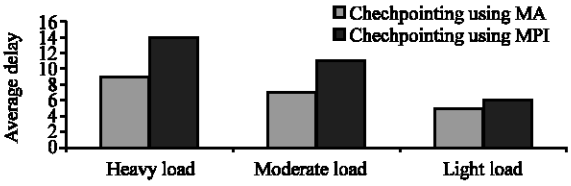

Fig. 11: The average delay under varying system loads

load but individual site utilization depends upon specific job migration scheme and the algorithm used for job migration.

The high level job migration provided by the agent system delivers basic load balancing across the resources in grid computing. Figure 10 shows the % of resource usage using check pointing with MA and MPI under heavy, moderate and light load. As can be expected, resource utilization increases as the system load increases. But what is important is that MA improves the resource utilization 20% in light load, 25% in moderate load, 30% in heavy load.

Figure 11 shows a comparison of average delay (seconds) using check pointing with MA and MPI under varying load. We use the average delay (μ) as an additional measure of QoS. As shown in Fig. 11, using a MA has a positive effect on μ under varying load condition. When the load and submission rate is high (1000 request at 5 per seconds) then check pointing with MA generates less delay compared to MPI. MA is also effective in light load (200 requests are sent per second) and generates low delay.

Figure 12 shows the comparison of execution time (seconds) using random selection of resources and MA using GA approach. Random Selection is that where user randomly selects the resources for job execution. Figure 12 clearly shows that MA using GA approach outperforms the random selection of resources in terms of execution time. The total execution time of job depends upon longest execution time of job so it is important that longest execution time of the job is minimized.

Figure 13 compares the execution time (seconds) and migration overhead involved under varying load. MA approach using check pointing generates low migration

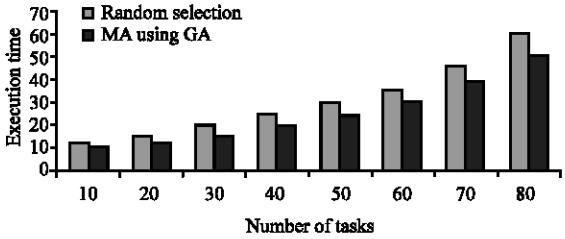

Fig. 12: Comparison of execution time using random selection and MA using GA

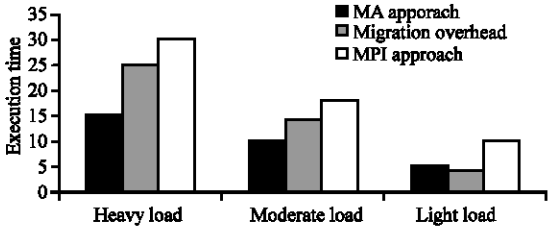

Fig. 13: Comparison of execution time using MA and check pointing and associated migration overhead

overhead compared to MPI. Because a lot of time will be consumed in writing, reading, transfer and executing the check points which will increase the total execution time.

RELATED WORKS

Load balancing is indispensable for a grid system to assure even distribution of workload on each node in grid. But one of the most difficult problems that arise on grid system is the selection of an efficient load balancing policy. The load balancing policy should be for evenly utilized grid and a minimum response time for the processed requests. In recent times grid computing has emerged as the attractive computing paradigm for solving a lot of computation intensive applications. But best solution to any computing problem is the execution of job with optimal resource usage and without fault. The fault tolerance is the incapability of communicating with other nodes due to hardware or system failure or extremely loaded situation. Past solutions (Knop *et al*., 1996; Guerraoui and Schiper, 1997) to fault tolerance are based upon check pointing. Checkpoint states usually include a large quantity of low level information such as stack and heap contents. This method is transparent but does not support portability for heterogeneous systems. For high-level check pointing the portability should be very high because only relevant application level program states will be transferred.

There are various approaches to make Grid computing fault tolerant (Lyu, 1995; Weissman, 1999; Wang *et al*., 1995). The basics, however, are checkpoint recovery and task replication. The former is a common method for ensuring the progress of a long-running application by taking a checkpoint, i.e., saving its state on permanent storage periodically. A checkpoint recovery is an insurance policy against failures. In the event of a failure, the application can be rolled back and restarted from its last checkpoint-thereby bounding the amount of lost work to be re-computed. Task replication is another common method that aims to provide fault tolerance in distributed environments by scheduling redundant copies of the tasks, so as to increase the probability of having at least a simple task executed. A brief overview of the options in fault tolerant computing on the Grid can be found in (Wang *et al*., 1995). There has been a variety of implementations that have addressed the problem of fault tolerance in Grid and distributed systems. (http.//wwwfp.globus.org/hbm/., 1999) provides a heartbeat service to monitor running processes to detect faults. The application is notified of the failure and is expected to take appropriate recovery action. Legion (Nguyen and Grimshaw, 1998; Chapin *et al*., 1999) provides mechanisms to support fault tolerance such as check pointing. Other grid systems like Netsolve (Casanova *et al*., 1998; Plank *et al*., 1999; Grimshaw *et al*., 1996; Gartner, 1999) have their failure detection and failure recovery mechanisms. They provide a single user-transparent failure recovery mechanism (e.g., re-trying in Netsolve and in Condor-G; replication in Mentat). FATCOP (Chen, 2001) is a parallel mixed integer program solver that works in an opportunistic computing environment provided by the Condor resource management system, using an implementation of a branch-and-bound algorithm. Peer-to-peer (P2P) systems also follow various methods for fault tolerance in their operation. An interesting overview of P2P systems is presented in (Zhuge *et al*., 2005; Lv *et al*., 2002).

For scheduling in grid, there are several systems that have been developed. The most significant attempts can be found in meta-schedulers such as Nimrod-G (Abramson *et al*., 1995; Abramson *et al*., 2002) software execution environments such as GRADS (Berman *et al*., 2001) and task brokers such as Condor-G (Frey *et al*., 2002). The latter is a product of a much more complicated entity that consolidates scheduling policies which comprised specialized workload management systems. Additionally, AppLeS (Faerman *et al*., 2003) is a scheduling system which primarily focuses on developing scheduling agents for individual applications on production. Other interesting works on scheduling and meta-scheduling are presented in (Weng *et al*., 2005) and (Subramani *et al*., 2002) where, in the former, the authors present a heuristic scheduling of bag-of-tasks with QoS constraints, while the latter handles the problem of distributed job scheduling in grids using multiple simultaneous requests. However, in coherent, integrated grid environments (such as Globus Project and Unicore (http.//www.unicore.org/forum.htm.) there are also scheduling and resource management techniques applicable in a more standard manner. Finally, other studies have also addressed resource management in grids such as the knapsack formulation problem (Parra-Hernandez *et al*., 2004). In this study the resource allocation in a grid environment is formulated as a knapsack problem and techniques are developed and deployed so as to maintain the QoS properties of a schedule and at the same time, to maximize the utilization of the grid resources.

CONCLUSION AND FUTURE WORK

In this study we have presented a architecture for fault tolerance and optimal resource utilization in grid computing under varying load conditions. The architecture consists of agents associated with their policies. The resource manager manages all resources in grid and uses genetic algorithm for optimal resource utilization. A fault detection and management service is provided so that submitted job is executed reliably and efficiently. FD detects the fault and FM tries to resolve it. GD is used for job migration with check pointing algorithm using MA. Also various agents communicate with each other using mobile group approach. Various metrics are used to discuss the results obtained including average response time, average wait time, grid efficiency, resource utilization under varying load. A comparison is made with respect to above defined metrics with check pointing using MA approach and MPI. Job migration overhead is discussed with respect to MA and MPI. In the future we would like to study the impact of time delay on these metrics and security associated when MA carries valuable information from one domain to another.

REFERENCES

Abramson, D. *et al*., 1995. Nimrod-G, A Tool for Performing Parametised Simulations Using Distributed Workstations. In Proceedings of the 4th IEEE Symposium on High Performance Distributed Computing, Virginia, pp: 112-121.

Abramson, D. *et al*., 2002. A computational economy for grid computing and its implementation in the Nimrod-G resource broker, Future Gener. Comput. Syst., 18: 1061-1074.

Al-Jaroodi, J. *et al*., 2003. A middleware infrastructure for parallel and distributed programming models on heterogeneous systems, IEEE Trans. Parallel and Distributed Systems, Special Issue on Middleware, 14: 1100-1111.

Anh, N.T., 2000. Integrating fault-tolerance techniques in grid applications, Ph.D. Thesis, University of Virginia.

Berman, F. *et al*., 2001. The GrADS Project. Software support for high-level grid application development. Int. J. High Perform. Comput. Applied, 15: 327-344.

Boloni, L. *et al*., 2006. Task distribution with a random overlay network, Future Gener. Computer Sys., 22: 676-687.

Braun, T.D. *et al*., 2001. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. J. Parallel Distributed Comput., 61: 810-837.

Buyya, R. *et al*., 2005a. The grid economy. Proceedings of the IEEE, 93: 698-714.

Buyya, R. *et al*., 2005b. Scheduling parameter sweep applications on global grids. A deadline and budget constrained cost-time optimization algorithm, Software. Practice Exp. J., 35: 491-512.

Casanova, H. *et al*., 1998. The GRID. Blueprint for a New Computing Infrastructure. IEEE Trans. Comput., 3: 159-180.

Chapin, S.J. *et al*., 1999. A. Grimshaw, Resource management in Legion. Future Gener. Comput. Syst., 15: 583-594.

Chau, K.W. *et al*., 2003. Knowledge-based system on optimum design of liquid retaining structures with genetic algorithms. J. Struc. Eng., ASCE, 129: 1312-1321.

Chau, K.W., 2004. A two stage dynamic model on allocation of construction facilities with genetic algorithms. J. Struc. Eng., ASCE., 129: 1312-1321.

Chen, Q. *et al*., 2001. FATCOP 2.0. Advanced features in an opportunistic mixed integer programming solver, Ann. Operational Res., 103: 17-32.

Cheng, C.T. *et al*., 2005. Multiple criteria rainfall-runoff model calibration using a parallel genetic algorithm in a cluster of computer. Hydrol. Sci. J., 50: 1069-1087.

Cheng, C.T. *et al*., 2006. Using genetic algorithm and TOPSIS for Xinanjiang model calibration with a single procedure. J. Hydrol., 316: 129-140.

Chess, D. *et al*., 1995. Itinerant agents or mobile computing. IEEE Personal Commun. Mag., 2: 34-49.

Dai, Y.S. *et al*., 2006. A grid-based pseudo cache solution for MISD biomedical problems with high confidentiality and efficiency. Int. J. Bioinform. Res. Appl., 2: 259-281.

Faerman, M. *et al*., 2003. Adaptive computing on the grid using AppLeS, IEEE Trans. Parallel Distrib. Syst., 14: 369-382.

Foster, C. *et al*., 1998. The Grid. Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers, Los Altos, CA.

Foster, C. *et al*., 2001. The anatomy of the grid enabling scalable virtual organizations. Int. J. Supercomputer Appl., 15: 60-74.

Foster, A. *et al*., 2000. A quality of service architecture that combines resource reservation and application adaptation. 8th International Workshop on Quality of Service, Italy, pp: 12-20.

Foster, C. *et al*., 2002. Grid services for distributed system integration. Computer, 35: 37-46.

Foster, C. *et al*., 2004. The Grid 2 Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers, Los Altos, CA.

Frey, J. *et al*., 2002 Condor-G. A computation management agent for multi institutional grids. Cluster Comput., 5: 237-246.

Gartner, F.C., 1999. Fundamentals of fault-tolerant distributed computing in asynchronous environments. ACM Comput. Surv., 31: 1-26.

Gergel, V.P. *et al*., 2005. Parallel computing for globally optimal decision making on cluster systems. Future Gener. Comput. Syst., 21: 673-678.

Grimshaw, A.S., 1996. Parallel Programming Using C++, pp: 382-427.

Imielinsky, T. and B.R. Badrinath, 1994. Wireless computing. Challenges in Data management. ACM Commun., 37: 18-28.

Jacob, B. *et al*. 2003. Enabling Applications for Grid Computing with Globus, IBM.Redbook.

Krauter, K., *et al*., 2002. A taxonomy and survey of grid resource management systems for distributed computing. Software-Practice Experience, 32: 135-164.

Li, C. and L. Li, 2004. Competitive proportional resource allocation policy for computational grid. Future Gener. Comput. Syst., 20: 1041-1054.

Li, H.X. *et al*., 2007. Parallel resource co-allocation for the computational grid. Computer Languages. Syst. Struct., 33: 1-10.

Lv, Q., *et al*., 2002. Search and replication in unstructured peer-to-peer networks. In: Proceedings of the 16th International Conference Supercomputing, pp: 84-95.

Lyu, M.R., 1995. Software Fault Tolerance. John Wiley and Sons, Chichester, UK.

Manivannan, D. and M. Singhal, 1999. Quasi-synchronous checkpointing: Models, characterization and classification. IEEE Trans. Parallel Distrib. Syst., 10: 703-713.

Manivannan, D. and M. Singhal, 2002. Asynchronous recovery without using vector timestamps. J. Parallel Distrib. Comput., 62: 1695-1728.

Marovi´c, B. and Z. Jovanovi´c, 2006. Web-based grid-enabled interaction with 3D medical data, Future Gener. Comput. Syst., 22: 385-392.

Meth, K.Z. and W.G. Tuel, 2000. Parallel checkpoint/restart without message logging. In: Proceedings of IEEE 28th International Conference on Parallel Processing (ICPP '00), pp: 253-258.

Nguyen-Tuong, A. and A.S. Grimshaw, 1998. Using reflection to incorporate fault-tolerance techniques in distributed applications. Computer Science Technical Report, University of Virginia, CS 98-34, 1998.

Parra-Hernandez, R. *et al*., 2004. Resource management and Knapsack formulations on the grid, in. Proc. of the 5th IEEE/ACM Int. Workshop on Grid Computing, GRID'04., pp: 541-552.

Parashar, M. *et al*., 2005. Application of grid-enabled technologies for solving optimization problems in data-driven reservoir studies, Future Gener. Comput. Syst., 21: 19-26.

Patel, R.B. and K. Garg, 2001. PMADE-A Platform for mobile agent Distribution and Execution, in Proceedings of 5th World Multi Conference on Systemics, Cybernetics and Informatics (SCI2001) and 7th International Conference on Information System Analysis and Synthesis (ISAS 2001), Orlando, Florida, USA., 4: 287-293.

Patel, R.B., 2004. Design and implementation of a secure mobile agent platform for distributed computing. Ph.D. Thesis. IIT Roorkee, India, 2004.

Patel, R.B. and K. Garg, 2004. A new paradigm for mobile agent computing. WSEAS Trans. Computers, 3: 57-64.

Patel, R.B. and K. Garg, 2005. A Flexible Security Framework For Mobile Agent Systems. Control and Intelli. Syst., 33: 175-183.

Plank, J.S. *et al*., 1999. Deploying fault tolerance and task migration with NetSolve, Future Gener. Comput. Syst., 15: 745-755.

Raimundo, J. *et al*., 2005. The mobile groups approach for the coordination of mobile agents. J. Parallel Distributed Computing, 65: 275-288.

Schlichting, R.D. and F.B. Schneider, 1985. Fail-stop processors: An approach to designing fault-tolerant distributed computing systems, in ACM Trans. Comput. Syst., 1: 222-238.

Schneider, J., 2003. Searching for Backbones-a high-performance parallel algorithm for solving combinatorial optimization problems, Future Gener. Comput. Syst., 19: 121-131.

Stelling, P. *et al*., 1998. A fault detection service for wide area distributed computations, in Proceedings of 7th IEEE Symposium on High Performance Distributed Computing, pp: 268-278.

Strom, R.E. and S. Yemini, 1985. Optimistic recovery in distributed systems, ACM Trans. Comput. Syst., 3: 204-226.

Subramani, V. *et al*., 2002. Distributed job scheduling on computational grids using multiple simultaneous requests. In: Proc. of the 11th IEEE International Symposium on High Performance Distributed Computing, Edinburgh, Scotland, pp: 359-367.

The Globus project, http.//www-fp.globus.org/hbm/.

The Unicore project, http.//www.unicore.org/forum.htm.

Vadhiyar, S. and J. Dongarra, 2003. A performance oriented migration framework for the grid, in. The Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, Tokyo, Japan, pp: 130-137.

Vraalsen, F. *et al*., 2001. Performance contracts Predicting and monitoring grid application behavior. Int. J. High Performance Comput. Appl., 15: 327-344.

Wang, F. *et al*., 1995. Determining redundancy levels for fault tolerant real-time systems. IEEE Trans. Comput., 44: 192-198.

Weissman, J.B., 1999. Fault tolerant computing on the grid. What are My Options, HPDC, pp: 351-352.

Weng, C. and X. Lu, 2005. Heuristic scheduling for bag-of-tasks applications in combination with QoS in the computational grid. Future Gener. Comput. Syst., 21: 271-280.

Yu, B. *et al*., 2005. Application of PGA on optimization of distribution of shopping centers. Lecture Notes in Artificial Intelligence, 3673: 576-586.

Zhuge, H. *et al*., 2002. A scalable P2P platform for the knowledge grid, IEEE Trans. Knowl. Data Eng., 17: 1721-1736.

Zomaya, A.Y., *et al*., 2001. An Introduction to Genetic-Based Scheduling in Parallel-Processor Systems, Solutions to Parallel and Distributed Computing Problems. Lessons from Biological Science, Zomaya, A.Y., F. Ercal and S. Olariu (Eds.), pp: 111-133, chapter 5. New York. Wiley.