A Generalized Approach for Fault Tolerance and Load Based Scheduling of Threads in Alchemi .Net

Vishu Sharma, Manu Vardhan, Shakti Mishra, Dharmender Singh Kushwaha
Department of Computer Science and Engineering
Motilal Nehru National Institute of Technology, Allahabad
Allahabad, India
Email: {cs0916, rcs1002, shaktimishra, dsk}@mnnit.ac.in

***Abstract*— Computational grids can be best utilized by the divide and conquer approach, when it comes to executing a large process. In order to achieve this, building multithreaded application is one of the efficient approaches. The threads are scheduled on different computational nodes for execution. One of the frameworks that support multithreaded applications is Alchemi, but it does not incorporate any load based scheduling and fault tolerance strategy. In Alchemi, a manager node uses first come first serve (FCFS) scheduling to schedule threads on executors (node that execute independent thread), but it does not consider any CPU load on which the executors are running. Moreover if an executor fails in between, then the manager node reschedules the thread on other executor node. One solution for the above problem is to save intermediate results from each thread and reschedule these threads on another executor. We propose an approach that provides fault tolerance in Alchemi by using Alchemi Replica Manager Framework (ARMF), where the manager node will be replicated on one of its executor node. The proposed algorithm is 6-16 percent more efficient than FCFS, when implemented in Alchemi.**

Keywords-ARMF; FCFS; fault tolerance; load based scheduling.

I. INTRODUCTION

A computational grid provides distributed environment in which user jobs can be executed either on local or on remote machines [2]. In grid, user jobs are considered as applications that contain the tasks to be executed. Further, each independent task is represented by a single thread. Whenever a user is having a job which contains multiple individual tasks it is better to use multithreading environment because thread creation and management is easier and faster than process creation. Threads provide following advantages over processes [20]:

- Thread creation takes less time because it uses the address space of process that owns it.
- Thread termination is easier than process
- There is less communication overhead between threads because address space is shared.

Figure 4 shows the architecture of Alchemi. It shows a manager connected with four executors. Alchemi provides API's that are used to create grid applications. In Alchemi, Gthread class is used to implement the multithreading [13]. Figure 1 shows the Gthread class and its structure. It contains an abstract method start ().Each thread is given a priority by a user. Alchemi .NET has the 5 priority levels from lowest to highest. Each application consists of several threads. The manager node is responsible for the scheduling of threads on different executors and collects the results from these executors after successful completion. The two issues related with Alchemi are scheduling of threads and fault tolerance.

The first issue is that of scheduling, where the manager node uses FCFS [17] policy for scheduling. It stores the threads according to their priority and schedules the highest priority thread on next available executor. It does not consider the CPU load of the processors on which the executors are running. If more than one executor is available at a time, it might happen that a thread is scheduled on a more loaded executor which can degrade the performance.

```
public abstract class GThread : MarshalByRefObject
{
public abstract void Start();
/* method is overridden by the class that inherits the
Gthread class*/ }
```

Figure 1. Structure of Gthread class.

Second issue is that of Fault tolerance, this helps system to recover from faults [4]. In case of Alchemi grid, if a thread is scheduled on an executor and due to some reasons, the executor crashes, the thread running on this executor also crashes. In such a case, the manager reschedules this thread on another executor and the thread is restarted from the scratch. Moreover there may be the case when the Alchemi manager can crash and all the executors currently registered with the manager will come to halt.

One solution to the above problem is discussed in [5]. The authors have used a file based implementation in which a file stores the intermediate results and if thread crashes it is rescheduled on another executor and resumes its execution from last successful result, without starting from the scratch. It reads the last successful result from the stored file.

The second limitation in [5] is that all the fault tolerance code overhead is on the user who submits the application. The Alchemi manager is not responsible for any kind of activity. Thus we came across the following issues that are yet to be resolved in Alchemi .NET.

- If a thread execution fails in between, then how the values produced by this thread (till the point of failure) can be saved at manager node and how the

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-153-3

remaining work of the failed thread is assigned to other thread. Approach given in [5] does not talk about how this kind of fault tolerance mechanism can be implemented in manager node. It completely relies on user. Neither have they discussed about the possibility of Alchemi manager failure.

- If the more than one executor is available at same time and the CPUs on which these executors are running might be overloaded then how to schedule threads to get a better solution.

To address above mentioned issues, a generalized approach is proposed as under, in which fault tolerance is provided for computational applications [12] running on a global grid.

- To provide a kind of check pointing scheme which stores the intermediate results produced by threads and the Alchemi manager node is incorporated with the facility to control the execution of failed threads and reschedule these threads on other available executors. In case of Alchemi manager failure the ARMF is proposed, which will provide the backup in such cases.
- To choose the best available executor on the basis of the load of CPU.

For more complex scientific application this approach may not work well as it requires users input. Hence, the proposed approach is confined to the computation intensive processes.

Rest of the paper is organized as follows. Section 2 describes the existing work done in fault tolerance and scheduling in grids. Section 3 shows the proposed approach. Section 4 shows the case study using the proposed algorithm and Section 5 derives the conclusion.

II. RELATED WORK

In load-based scheduling [18], load information can't be exchanged much frequently due to network communication overheads [2]. It is desirable to exchange the load information only when it is needed.

In a system, fault tolerance is achieved by means of some redundancy that could be hardware, software or time redundancy [19].

Vladimir et al. [7] discuss about the scheduling of divisible load applications, where the resources are selected dynamically, based on the intermediate results. In this approach, application specific requirement also plays a vital role in selecting the resources. But this approach is applied at application level and does not concentrate on multi-threaded grid [15] environment.

Zeljko et al. [8] discusses an improved scheduling strategy in Alchemi. This approach still relies on a static strategy for selecting the executors and adds nothing to fault-tolerance. To achieve fault tolerance, a file based technique is proposed in [5]. First problem with this approach is that it places the burden of creating and manipulating the file on the user who creates the application and the manager does not contribute in any kind of fault tolerance activity. Second problem is that for each thread there is a single file, means incurring more overhead on the manager node. This approach [5] has been shown only for one application. Authors have not discussed how other applications can be implemented using this approach.

One of the characterization techniques is given in [10]. In this technique, individual machine faults are defined as, resource level fault and faults in global environment of grid are considered as service level faults. This paper does not elaborate much about the resumption of jobs from the point where it was crashed.

Another improved approach is given in [11]. Fault tolerance is achieved at job level but as each job can be divided into individual tasks using multithreading so several issues like which thread got faulted, how to combine the results from faulted threads etc remain unhandled.

An approach for thread scheduling is shown in [16], where different threads are scheduled to download files from different servers. But in this approach if a thread fails to execute, it is rescheduled after all threads complete their execution.

All the above discussed literature work motivated us to put efforts for providing a novel solution to fault tolerance and load based scheduling in Alchemi .NET.

III. PROPOSED APPROACH

In our approach two concepts, first is fault tolerance and second is scheduling of threads, based upon CPU load are integrated into single algorithm. We first discuss about the fault tolerance approach followed by the thread scheduling based on CPU load. The proposed approach did not consider the manager load, as the thread will always execute on the executor node, not on the manager node. There may be the case of manager failure, which we have discussed below.

A. Fault Tolerance Approach

In Alchemi .NET the applications are divided into individual threads and these threads are scheduled on currently available executors. If a thread execution stops in between then the work done by that thread till that point will be lost.

In [5], an approach is proposed in which file is created for each thread which keeps track of thread execution. This approach puts extra burden of creating and using the file over the application programmer who creates the application.

We propose an approach that enhances this idea [5] by incorporating the manager with the capability of creating and maintaining the file. Each application, submitted by a different user is different and hence the intermediate results (variables) would be different. We try to generalize this approach so that different kind of applications can be executed in the same way. To support this kind of dynamicity, we are using the XML-file. As the application is submitted, the manger node creates an XML-file with relevant information loaded into it. This information is responsible for resuming a crashed thread.

A big challenge in this approach is how to identify these variables. In our approach these variables are supplied by the user who submits the application because the user knows what and where the values must be stored. During the thread

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-153-3

execution, the executor is responsible for saving these values into the XML-file that is on manager. Whenever a crashed thread is rescheduled on different executor the manager node will extract the values from that XML-file and will pass it to the thread so that it can resume its operations.

```
Public class table: Gthread /* user code */ {  table (
int starting_number, int last_number)
    {/* constructor initializes the values in XML file */
    /* initialization of values done by manager */
    }  Public void start()
    {for(num=starting_number;num<=last_number;
number++)
        for( int i=1; i<= 10; i++)
            { result=num*i; }
    savetofile(num, result);
    }}  Savetofile(values )/* method runs on executor */
    {  /* sends intermediate values to the manager node
*/ }
```

Figure 2. Proposed structure of thread implementation.

Figure 2 shows the structure of the threaded class that a user implements. This class extends the Gthread class given in Figure 1. The Structure of the XML file is given in Figure 3. This file contains the values for threads for which processing has been successful.

```
<file  application_id= " "><thread>
<init><thread_id> 123</thread_id>
<first number>1</first number>
<last number>5<last number>
<completed>yes</completed></init></thread>
<thread>
<init><thread_id>163</thread_id>
<first number>6</first number>
<last number>10<last number>
<completed>no</completed>
</init>
</thread> </file>
```

Figure 3. Structure of XML file.

In the existing file-based fault tolerance approach [5], fault tolerance is supported at user end. Fault tolerance is completely dependent on application user. In our proposed approach, fault tolerance is supported by the Alchemi manager, application user need not to concern about its implementation.

Next, in Alchemi architecture, there is no provision for handling the situation where manager can fail. Under these circumstances all the Executors registered with the failed manager will stop executing, and the whole system will come to halt. There should always be some backup / replica manager, so that single point failure can be avoided.

Alchemi manager which is responsible for managing the execution of grid applications can be replicated. This can be achieved by replicating the Alchemi manager at its one of the Executor, which is currently registered with this manager.

Figure 4 describes the whole scenario. The manager node is connected with four executors. Each executor executes an independent thread. User application is containing 3 threads.

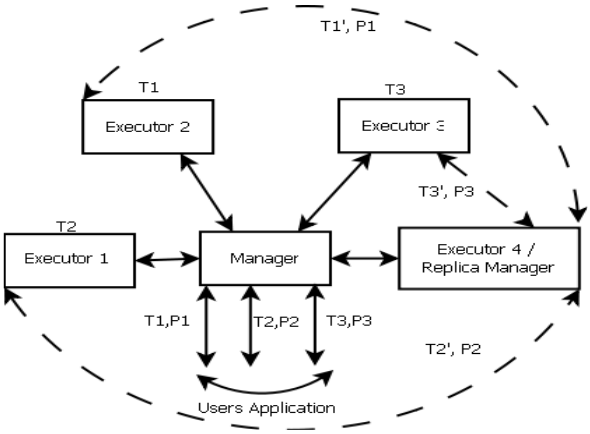

Figure 4. Architecture of Alchemi and Alchemi Replica manager.

P1, P2, P3 are the thread priorities assigned by the user for the respective thread. T1', T2', T3' are the thread associated with the Replica manager which is on Executor 4.

The information that needs to be transferred to the Executor node, so that the Alchemi manager can continue functioning from the point of failure and not from the scratch, is stored in a XML file with the manager. This XML file needs to be replicated to that Executor node, which is acting as a replica of Alchemi manager. Periodic updation of this XML file is required, so as to maintain the consistency of the system.

The information that needs to be transferred to the executor node is stored in a XML file with the manager, so that the Alchemi manager can continue functioning from the point of failure and not from the scratch. This XML file needs to be replicated to that Executor node, which is acting as a replica of Alchemi manager. Periodic updation of this XML file is required, so as to maintain the consistency of the system.

In the present Alchemi framework, an executor can register itself only with one manager. Issue associated here, from the developers/programmers perspective is "how the Executor will register itself with the new manager i.e., the replicated manager in case of manager failure". With the present framework, if the manager fails, the new replica manager needs to inform all the executors, registered with the failed manager, to get them registered with the new replica manager. Or there should be some provision by which an executor can register it with more than one manager.

B. Modified Scheduling Algorithm

Alchemi .NET provides its grid API that is used to develop grid applications to be submitted to the Alchemi. Each application contains threads. Number and priority of

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-153-3

threads are defined by the application programmer. It does not consider current performance of the CPU on which the executor is running. If at the same time two executors are available and one of these is overloaded whereas other is not, so it might happen that a highest priority thread is scheduled on an executor that is overloaded. In those cases when the higher priority thread execution duration is large, this overloaded executor might degrade the performance.

In the proposed approach, an executor does not send its load information periodically, rather it sends it whenever an executor finishes execution of a thread and it is ready to receive a new thread from the manager. We assume that no thread is interrupted during its execution due to the load information on its machine.

In Figure 5 default mechanism of selecting the executors is shown.

```
Step1: Thread=gethighestprioritythread();
Step2: Executor=Getnextavailableexecutor()
Step3: create new schedule with executor and thread.
Step4: Schedule(dedicateschedule);
```

Figure 5. Default scheduling mechanism in Alchemi.

Figure 6 shows the modified algorithm, if more than one executor is available at the same time our algorithm selects the best one.

```
Step1: Thread= Gethighestprioritythread();
Step2: Execut_available[]=Getcurrent_avail_executor()
       Executor= Executoravailable[].getleastloaded().
Step3: Create new schedule with executor and thread.
Step4: schedule(dedicateschedule);
```

Figure 6. Modified mechanism.

C. Algorithm

The algorithm combines both the approaches discussed above. Its theoretical description is given in Figure 7.
The architecture of the proposed approach is shown in Figure 8. A ft_thread is added at manager and executor nodes. At manager node the ft_thread is running continuously and is responsible for receiving the intermediate values from the ft_thread running on executors. It writes the intermediate values into the XML file and reads them in case a faulty thread needs to be rescheduled.

1. Get the highest priority thread from the database.
2. Create the entry in XML file for that thread.
3. Get the available executors check their load factor and if more than one executor is available get the minimally loaded executor.
4. Receive the intermediate values sent by the executor for that thread.
5. Replace the existing value in XML file with the recently received values.
6. If executor gets disconnected then check the thread status allocated to that executor. If it is not completed create new thread with the same thread id that was executing on the crashed executor.
7. Supply the last successful results to that newly created thread so that it can resume its execution.
8. Get the minimally loaded executor and assigned this thread to that executor.
9. Repeat steps 1 to 8 until the thread database is empty.

Figure 7. Proposed algorithm.

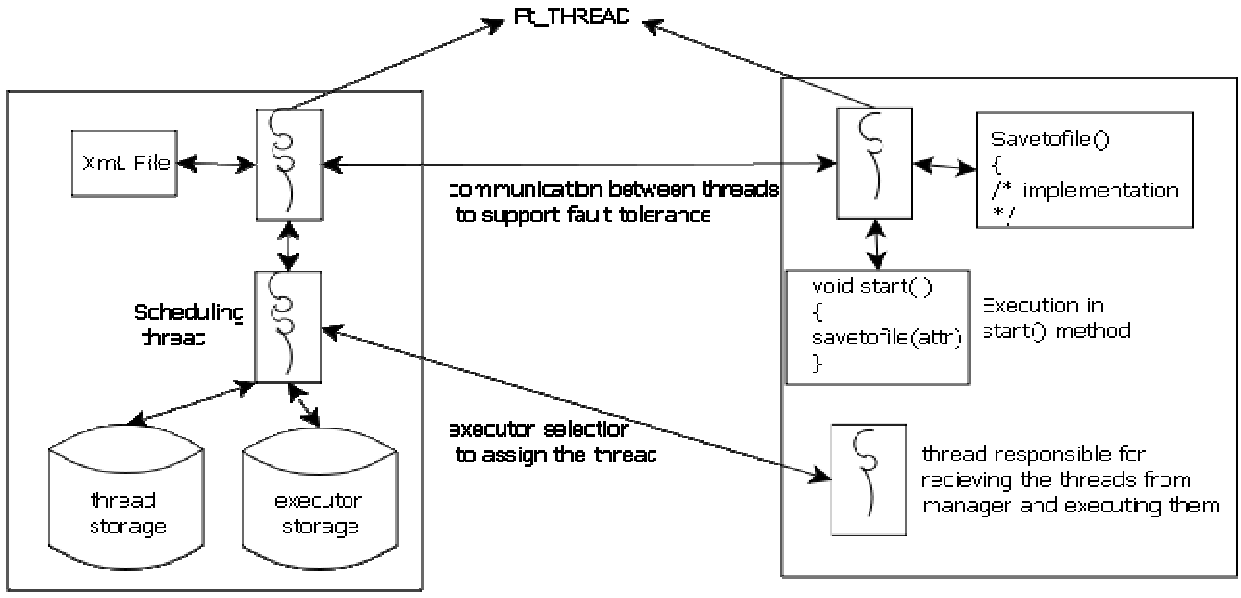

Figure 8. Architecture of Fault Tolerant Alchemi.

IV. CASE STUDY

We evaluate the scenario where an overloaded executor might be a bottleneck for the performance. In Figure 9, we show an example with **t**hree executors on which threads are scheduled. We assume that all executors that are not overloaded execute the threads in approximately same time.

In Figure 9, an executor is marked as overloaded and it takes more time to execute a thread as compared to an average loaded or underloaded executor.

An average loaded or underloaded executor takes 4 units of time to execute a high priority thread and 2 units of time to execute a low priority thread whereas an overloaded executor takes 6 units of time for high priority thread and 3 units of time for low priority thread. Hence the completion time for this application according to FCFS scheduling is 9 units of time.

In Figure 10, we see another arrangement of threads on the executors. In this low priority threads are scheduled on overloaded executor and all high priority threads are scheduled on less overloaded executors. The completion time of the application is 8 units of time.

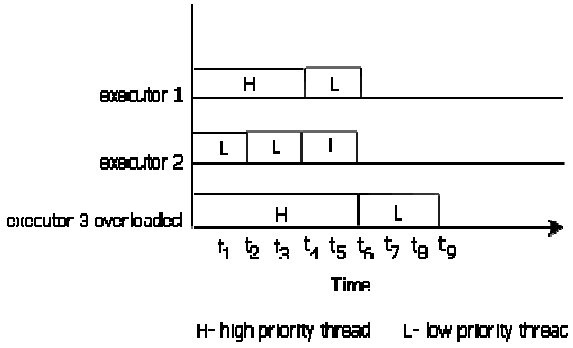

Figure 9. Arrangement of threads on executors according to default mechanism.

Load information collected from the executor also helps in selecting the best available executor whenever a thread is rescheduled after a crash. In our approach we assume that if at any point of time two executors are available we select one which is less loaded.

In the simulated environment we analyze the behavior of proposed application with different applications. These applications are included in random. In Alchemi, different executor nodes are connected to manager node. From these available executor nodes some are overloaded in comparison to others.

Table I shows five applications, number of high and low priority threads for each application. In this table, column name A.N. stands for application number, N.T. for Total number of threads in an application, N.H.P for Number of high priority threads, N.L.P. for Number of low priority threads and E.E.T. for Expected execution time on normal executor. In Table II, completion time for FCFS and proposed algorithm is shown. The total number of threads in a single application is shown in Table I. The execution time for a thread is shown on a normal executor. We assume that an overloaded executor takes 50% more time to execute a thread. In Table I application number 4 has threads of same type, i.e., all the threads are having same priority. In this case also, our proposed algorithm performs well.

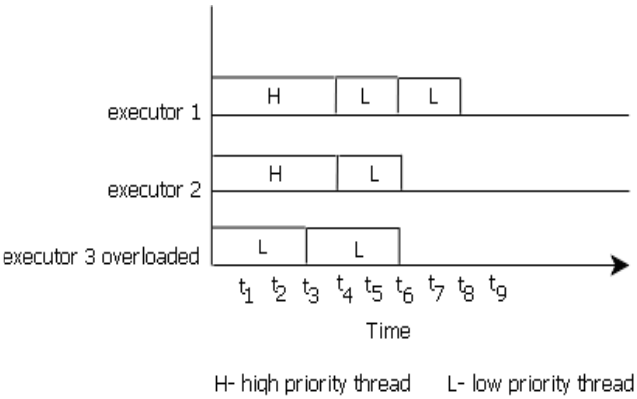

Figure 10. Arrangement of threads on executors according to proposed algorithm.

TABLE I. APPLICAION CHARACTERISTICS. H RPRESENTS THE HIGH PRIORITY THREAD AND L REPRESENTS THE LOW PRIORITY THREAD

A.N.	N.T.	N.H.P	N.L.P	E.E.T.	
				H	*L*
1	7	2	5	4	2
2	14	2	12	6	4
3	11	2	9	10	6
4	9	9	0	6	-
5	6	4	2	10	5

Figure 11 shows the results obtained from FCFS and proposed algorithm in simulated environment. It shows that our proposed algorithm gains better completion time. Figure 8 also shows that for a given application set, our proposed algorithm is 6-16 % more efficient in comparison to FCFS algorithm. In case where all the threads have same priority, it is 11% more efficient than the FCFS algorithm.

V. CONCLUSION

An approach that achieves fault tolerance supported by manager node of Alchemi is presented in this paper. In comparison to other approaches, the scheduling of threads on various nodes after the crash requires no user intervention. Rather the proposed approach implements fault tolerance in system by using manager node and executor node. We also propose an Alchemi Replica Manager Framework (ARMF) and a scheduling algorithm based on the load information of executor nodes. ARMF replicates the XML-file, which is maintained by the manager node and stores all the required information about the threads executing on the executors, to one of its executor, which will be acting as the replica manager in case of manager failure. Our proposed algorithm selects the executors depending upon the load information of currently available executors. This helps Alchemi manager to select best executor (least loaded for a high priority

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-153-3

thread) amongst available ones. In performance study, it has been found that the proposed approach is 6 – 16 % more efficient than FCFS, when implemented in Alchemi. Alchemi Replica Manager Framework (ARMF) provides a mechanism to replicate manager node to one of its executor.

TABLE II. COMPARISON BETWEEN PROPOSED ALGORITHM AND FCFS

Application number	No.of exec-utors	No.of over loaded CPUs	FCFS Comp-letion time	Proposed algorithm completion time
1	3	1	9	8
2	4	2	21	18
3	3	1	33	28
4	3	1	18	16
5	3	1	22.5	20

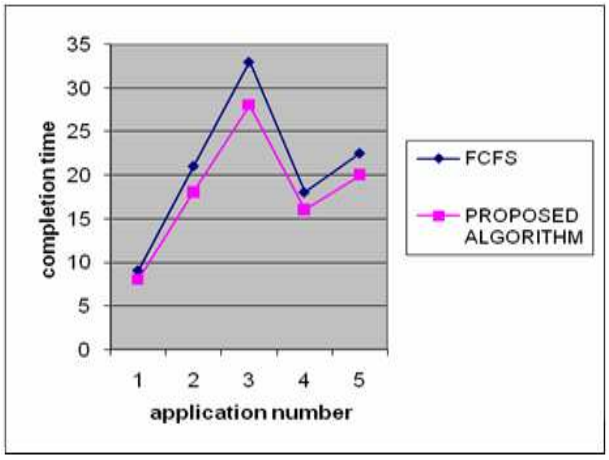

Figure 11. Performance study of both algorithms.

REFERENCES

[1] Sunita Bansal, Gowtham K, and Chittrnjan Hotta: Novel adaptive scheduling Algorithm for computational grids. Proceeding IMSAA'09 Proceedings of the 3rd IEEE international conference on Internet multimedia services architecture and applications, pp. 1-5, 2009.

[2] Ruchir Shah, Bhardwaj Veeravalli, and Manoj Misra: On the Design of Adaptive and Decentralized load balancing algorithms with Load estimation for computational grid environments, IEEE transactions on parallel and distributed systems, vol. 18, no. 12, pp. 1675-1685, 2007.

[3] Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal: Alchemi: A .NET-based Grid omputing Framework and its Integration into Global Grids. In: Grid Computing and Distributed **S**ystems (GRIDS), Technical Report, GRIDS-TR-2003-8, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, pp. 1-17, 2003

[4] William C. carter: Fault-Tolerant Computing: An Introduction and a Viewpoint, IEEE TRANSACTIONS ON COMPUTERS, vol. C-22, no. 3, pp. 225 – 229, 1973.

[5] Md. Abu Naser Bikas, AltafHussain, Abu Awal Md. Shoeb, Md. Khalad Hasan, and Md. Forhad Rabbi: File Based GRID Thread Implementation in the .NET-based Alchemi Framework, Multitopic ConferenceI, NMIC. IEEE Intern., pp. 468-472, 2008.

[6] Veeravalli Bharadwaj, Debashish Ghose, and Thomas G. Robertazzi: Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems, Cluster Computing 6, pp. 7–17, 2003, 2003.

[7] Vladimir V. Korkhov, Jakub T. Moscicki, and Valeria V. Krzhizhanovskaya: The User-Level Scheduling of Divisible Load Parallel Applications With Resource Selection and Adaptive Workload Balancing on the Grid, IEEE systems journal, vol. 3, no. 1, pp. 121-129, 2009.

[8] Zeljko Stanfel, Goran artinovic, and ZeljkoHocenski: Scheduling Algorithms for Dedicated Nodes in Alchemi Grid. IEEE International Conference on Systems, Man and Cybernetics, pp., 2531 – 2536, SMC 2008.

[9] Gracjan Jankowski, Radoslaw Januszewski, and Rafal Mikolajczak.: Improving the fault-tolerance level within the GRID computing environment - integration with the low-level checkpointing packages, CoreGRID Technical Report Number TR-0158, June 16, 2008.

[10] Jes´us Montes CeSViMa, Alberto S´anchez, and Mar´ıa S. P´erez.: Improving grid fault tolerance by means of global behavior modeling, Ninth International Symposium on Parallel and Distributed Computing, pp. 101-108, 2010.

[11] HwaMin Lee1, DooSoon Park1, Min Hong1, Sang-Soo Yeo2, SooKyun Kim3, and SungHoon Kim4.: A Resource Management System for Fault Tolerance in Grid Computing, International Conference on Computational Science and Engineering, pp. 609-614, 2009

[12] Nirmalya Roy and Sajal K. Das: Enhancing Availability of Grid Computational Services to Ubiquitous Computing Applications, IEEE transactions on parallel and distributed systems, vol. 20, no. 7, pp. 953-967, 2009.

[13] Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal: Alchemi: A .NET-based Enterprise Grid Computing System, 6th International Conference on Internet Computing. Las Vegas, pp. 1-10, 2005.

[14] Sandeep Singh Rawat and Dr. Lakshmi ajamani: Experiments with CPU Scheduling Algorithm on a Computational Grid, IEEE International Advance Computing Conference (IACC 2009), pp. 71-75, India, 2009

[15] Jos´e Augusto Andrade Filho, Rodrigo ernandes de Mello, and Evgueni Dodonov: Toward an efficient Middleware for Multithreaded Applications in Computational Grid, 11th IEEE International Conference on Computational Science and Engineering, pp. 147-154, 2008.

[16] Suvarna N. A and Dinesh Chandra: Evaluation of Improvement Algorithms for dynamic Co-Allocation with respect to parallel downloading in Grid Computing, First International Conference on Integrated Intelligent Computing, pp. 79-83, 2010.

[17] U. Schwiegelshohn and R. Yahyapour: Analysis of first-come-first serve parallel job scheduling, Proceedings of the ninth annnal ACMSIAM symposium on Discrete algorithms (SODA'98), pp. 629-638, 1998.

[18] Cui Zhendong and Wang Xicheng.: A Grid Scheduling Algorithm Based on Resources Monitoring and Load Adjusting, Knowledge Acquisition and Modeling Workshop, 2008, KAM Workshop, pp. 873-876, 2008.

[19] Nils Mullner, Abhishek Dhama, and Oliver Theel: Deriving a Good Trade-off between System Availability and Time Redundancy, Symposia and Workshops on Ubiquitous, Autonomic and Trusted Computing, pp. 61-67, 2009.

[20] http://www.personal.kent.edu/~rmuhamma/OpSystems/Myos/threads.htm accessed on 10-05-2011.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-153-3