A novel Task Replica based Resource Scheduling Algorithm in Grid computing

N. Kiran, V. Maheswaran, M. Shyam and P. Narayanasamy
College of Engineering, Guindy (CEG), Anna University, Chennai

Abstract— *Increased network bandwidth, more powerful computers and the Internet have driven the on-going demand for new and better ways to compute. Grid computing allows one to unite pools of servers, storage systems, and networks into a single large system so you can deliver the power of multiple-systems resources to a single user point for a specific purpose. A primitive algorithm on scheduling process of grid computing has been studied. Based on it, a makespan algorithm has been proposed using the novel idea of process replication. In this algorithm we have made full utilization of the available resources and mathematically obtained a TPCC which is only slightly higher than the TPCC of the ideal parallel and distributed computing systems.*

Index Terms — *Grid Computing, TPCC, Makespan, Task replica.*

I. INTRODUCTION

1.1 Grid Computing

Grid computing [2] allows us to unite pools of servers, storage systems, and networks into a single large system so you can deliver the power of multiple-systems resources to a single user point for a specific purpose. To a user, data file, or an application, the system appears to be a single enormous virtual computing system.

With grid computing, an organization can transform its distributed and difficult-to-manage systems into a large virtual computer that can be set loose on problems and processes too complex for a single computer to handle efficiently. The problems to be solved can involve data processing, network bandwidth, or data storage. The systems linked in a grid might be in the same room or distributed around the world. They might be running different operating systems on many hardware platforms. They might even be owned by different organizations. Regardless of the depth of a grid's resources, all the grid user experiences is the processing resources of a very large virtual computer.

N. Kiran is with the Department of Computer Science and Engineering, College of Engineering, Guindy, Chennai – 600025, INDIA (e-mail: nkiran87@yahoo.com)
V. Maheswaran is with the Department of Electronics and Communication Engineering, College of Engineering, Guindy, Chennai – 600025, INDIA (e-mail: kvmakes@yahoo.com)
M. Shyam is with the Department of Electronics and Communication and Engineering, College of Engineering, Guindy, Chennai – 600025, INDIA (e-mail: scintillatingstuffs@yahoo.co.in)
P. Narayanasamy is with the Department of Computer Science and Engineering, College of Engineering, Guindy, Chennai – 600025, INDIA (e-mail:sam@annauniv.edu)

1.2 Basic Grid Computing Architecture

Practical grids [1] are generally described in terms of layers as shown in Figure 1. The lowest layers (the 'platform') comprise the hardware resources, including computers, networks, databases, instruments, and interface devices. These devices, which will be geographically distributed, may present their data in very different formats, are likely to have different qualities of service (e.g. communication speeds, bandwidth) and are likely to utilize different operating systems and processor architectures.
A key concept is that the hardware resources can change over time - some may be withdrawn, upgraded or replaced by newer models, others may change their performance to adapt to local conditions - for example restrictions in the available communications bandwidth.

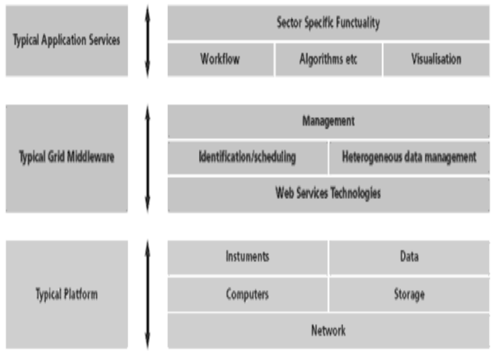
Typical Application Services
Sector Specific Functuality
Workflow
Algorithms etc
Visualisation
Typical Grid Middleware
Management
Identification/scheduling
Heterogeneous data management
Web Services Technologies
Typical Platform
Instuments
Data
Computers
Storage
Network

Fig. 1: Grid Architecture

The middle layers (sometimes referred to as 'middleware') provide a set of software functions that 'buffer' the user from administrative tasks associated with access to the disparate resources. These functions are made available as services and some provide a 'jacket' around the hardware interfaces, such that the different hardware platforms present a unified interface to different applications. Other functions manage the underlying fabric, such as identification and scheduling of resources in a secure and auditable way. The middle layer also provides the ability to make frequently used patterns of functions available as a composed higher-level service using workflow techniques.

1.3 Scheduling

Scheduling [3] is a key concept in computer multitasking and multiprocessing operating system design, and in real-time operating system design. It refers to the way processes are assigned priorities in a priority queue. This assignment is carried out by software known as a scheduler. Operating systems may feature up to 3 distinct types of schedulers: a long-term scheduler (also known as an admission scheduler), a mid-term or medium-term scheduler and a short-term scheduler (also known as a dispatcher).

Scheduling algorithm is the method by which threads or processes are given access to system resources, usually processor time. This is usually done to effectively load balance a system. The need for a scheduling algorithm arises from the requirement for most modern system to perform multitasking, or execute more than one process at a time. Scheduling algorithms are generally only used in a time slice multiplexing kernel. The reason is that in order to effectively load balance a system the kernel must be able to forcibly suspend execution of threads in order to begin execution of the next thread.

II. SCHEDULING METRICS

We are considering a primitive algorithm aimed at reducing the Makespan. We have used this algorithm as the basis for the algorithm we have constructed.

2.1 Basic Terminologies

The **length L** of a task is the number of instructions in the task. The **speed** of a processor is the number of instructions computed per unit time. A grid is heterogeneous, so processors in a grid have various speed by nature. In addition, the speed of each processor varies over time due to the load by the original users in public-resource computing. That is, the speed of each processor is the excess computing power of the processor which is not used by the original users and is dedicated to a grid. Let s(p,t) be the speed of processor p during time interval [t,t+1] where t is a non-negative integer. Without loss of generality, we assume that the speed of each processor does not vary during time interval [t,t+1] for every t by adopting enough short time as the unit time. We also assume that we cannot know the value of any s(p,t) in advance. s(p,t) may be zero if the load by the original users is very heavy or the processor is powered off. For simplicity, processor addition, processor deletion, and any failure are not considered here.

2.2 Total Processor Cycle Consumption (TPCC)

With the above constraints in a grid computing system, we can be sure that the TPCC of a grid is always greater than parallel and distributed computing systems [5]. The TPCC of parallel or distributed computing system is always going to be nL, Where n is the total number of tasks that are to be fed to the grid. How far the is TPCC of our algorithm from the ideal case of parallel and distributed computing system is our subject of interest. In this work we finally prove that the TPCC is going to be just a logarithmic factor higher than the ideal case. We shall show here that the TPCC of parallel and distributed computing system is nL. In parallel and distributed computing systems the total processor allocation or the speed as defined above is known earlier before the tasks are fed to the grid. Hence the speed is not a dynamically varying parameter [6]. Hence it is always possible to predict the TPCC even before the tasks are fed to the grid.

III. THE PRIMITIVE ALGORITHM

3.1 A Schedule

Let T be a set of n independent tasks with the same length L. Let m be a number of processors in a computational grid. A **schedule S** [2] of T onto a grid with processors is a finite set of triples (v,p,t) and t is the **starting time** of task v. A triple (v,p,t) means that the processor p computes the task v between time t and time t+d where d is defined so that the number of instructions computed by the processor p during the time interval [t, t+d] is exactly L. We call t+d the **completion time** of the task v. The starting time and completion time of a task are not necessarily integral, though the processor cycles (length of the task) is always integral.

3.2 Algorithm

Step 1: Assign the first m tasks the first m processors.
Step 2: Immediately after a processor becomes free, assign the next task to that processor
Step 3: Repeat the above step until all the tasks are completed
Step 4: Makespan is the time of completion of the last task
Step 5: And TPCC is total number of instructions that the processors are capable of executing till the Makespan.
The above algorithm is better understood with the help of Figure 2.

3.3. Example

Here we consider a set of five tasks to be completed. Let us assume that the task scheduler in its random brokering according to the speed of computation allots these five tasks to a computing element of the grid which has three processors. These three processors are variable speeds which can be known only after the execution of the tasks. The speeds which are found out after all the tasks get over are shown in the Table 1. Processor 1 at the first second (for simplicity sake lets us consider one time duration as one second) can complete a task of length 2, similarly processors 2 and 3 can complete tasks of lengths 5 and 10 respectively in the first second. This table data is purely random and varies from instant to instant. This behavior is due to the fact that only excess resources are given to the grid for usage.

Now once the tasks are fed to a computing element, as per the algorithm the first m tasks are fed to the available m processors. So, in our example the first three tasks are fed to the first 3 processors. Task 2 gets completed quickly due to its greater resource allocation to the grid at that particular period of time. Immediately after task 2 gets completed, the next task in the queue V4 is given to processor 2. Among all running tasks, V3 completes next in processor 3 and hence task V5 is allotted to processor 3. The schedule of T is shown in Table 1.

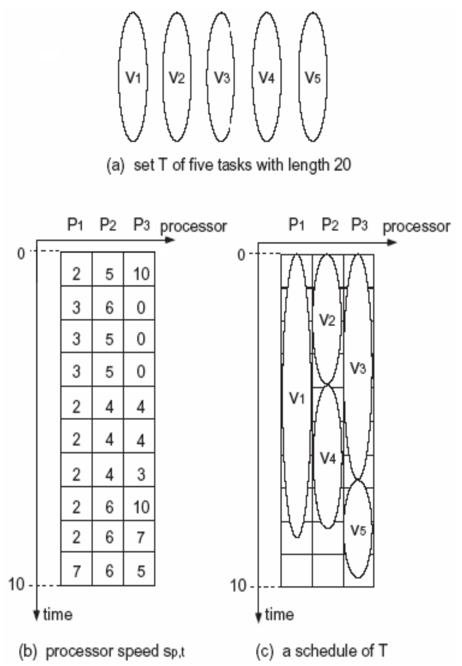
(a) set T of five tasks with length 20
V1 V2 V3 V4 V5
P1 P2 P3 processor
0
10
time
(b) processor speed $s_{p,t}$
(c) a schedule of T

Fig. 2: Scheduling through primitive algorithm

Tasks	The schedule	Explanation
V1	$(v_1,p_1,0)$	Task V1 is started at time 0s in processor p1
V2	$(v_2,p_2,0)$	Task V2 is started at time 0s in processor p2
V3	$(v_3,p_3,0)$	Task V3 is started at time 0s in processor p3
V4	$(v_4,p_2,19/5)$	Task V4 is started at time 19/5s in processor p2
V5	$(v_5,p_3,20/3)$	Task V5 is started at time 20/3s in processor p3

Table1. Schedule of Tasks

IV. TASK REPLICA ALGORITHM

4.1 Task Replica Algorithm

In this section, dynamic scheduling algorithm mentioned above is used as reference. If any one of the m tasks gets completed, then the algorithm replicates the earliest uncompleted arrived of the m tasks into the processor of the task which completed at that instance of time.

If any of the task v of processor p gets completed, then the algorithm saves the result of the task v and terminates any existing replicas of v. The figure 3 explains how the proposed algorithm works in case of 4 processor and 7 available tasks.

As shown in figure 3, the computing element in this example has 4 processors and seven tasks in the queue which are to be executed by the available processors. The initial allocation to the processor proceeds the same way as the primitive algorithm. Initially, the first four task are allotted to the available 4 processors. Then task V2 gets completed and task V5 is allotted to processor 2. Then V4 in processor 4 gets completed and the next task in the queue V6 is fed to processor 4. Then V5 in processor 2 gets over and task V7 is allotted there. Thus far, this algorithm has proceeded the same way as in the primitive algorithm. Now when a task gets over, this algorithm replicates or duplicates the first task fed to the grid that still remains incomplete.

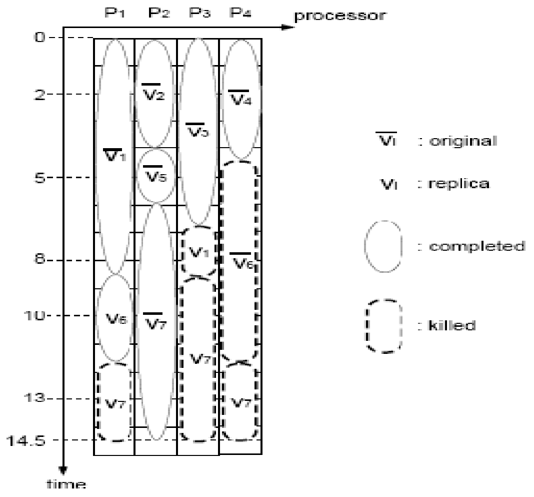
P1 P2 P3 P4 processor
0
2
5
8
10
13
14.5
time
$\overline{V_1}$ $\overline{V_2}$ $\overline{V_3}$ $\overline{V_4}$ $\overline{V_5}$ $\overline{V_6}$ $\overline{V_7}$
V1 V6 V7
$\overline{V_i}$: original
V_i : replica
: completed
: killed

Fig. 3: Scheduling process of Proposed Algorithm

4.2 Replication and Termination

In our case, when V3 gets over, the scheduler is left with the job of replication. The scheduler decides on the task to be replicated based on the first job fed to the grid that still remains incomplete. In our case, V1 which is the first task fed to the grid still remains incomplete. So, the scheduler replicates a copy of V1 in processor 3 when V3 finishes. In the figure 3, the replicas are given by ordinary variables and their corresponding replicas given by complemented variables. Then the task V1 gets completed in processor P1 itself . In this case, the scheduler kills or terminates the replica. By this way, we make sure that there is no wastage of processor time due to the replicas created. In case the replica is completed first, the scheduler will kill or terminate the original process wherever it is at that point in time.

Now, the scheduler has two free processors 1 and 3. It feeds the first fed still incomplete task V6 to the first processor 1 and the next incomplete task in the order of priority to processor 3. In our case it is task V7. So, task V7 is fed to processor 3. Now the original task V6 is completed in processor 4, hence the replica in processor 1 is terminated immediately. Again we have 2 free processors now (Once the replication process starts the number of processors available is always greater than 2). Now we are left with the only incomplete task V7. This task is fed onto both the available free processors 1 and 4. Note that the last task runs in all the available processors after the last but one task gets completed. Now if one of these four gets completed all the others are killed.

These inferences can be drawn from the algorithm.

i. In the primitive algorithm we have considered, we find that the resource remains unused for a long time. During the course when the resource remains idle, the local use of the resource may get reduced drastically and it may produce huge processing capability, which goes unused [4]. The algorithm we have suggested makes a good use of the sudden rise in resource processing capability.

ii. Replicas are created only for last (m-1) that are put into the processors for scheduling.

iii. At any time instant, the difference of the size between every pair of replica groups is at most one.

iv. We find that at no point of time, the processors remains idle, where as in the previous case, during the execution of last m task, some processors remains idle, until the rest of the tasks are completed.

V. MATHEMATICAL ANALYSIS

The observations in Section IV lead to the following mathematical bases for grid computing.

5.1 Calculation of TPCC

For calculating TPCC, we have to calculate the maximum number of instructions that will be executed in the grid. This will include the original tasks replicas created if any.

5. 2 Finding the total number of replicas

For finding the total no. of replicas we need to know the total number of tasks that are actually getting replicated. Observation 2 helps us in finding that.

Observation 2 follows from the task that replicas are created only after the completion of (n-m) tasks and the next task. Therefore the replicas are created only for the last (m-1) tasks. For example, in a 9 task, 4 processor system initially it has to put in grid all the 9 tasks before creating replicas. That can be possible only if 5 tasks are fully complete. Then for the replica of the first process to be created another one task has to get over. Therefore it is impossible to create replicas for first 6 tasks. In other words, replicas are created only for m-1 last tasks (in this case 3 last tasks).

Observation 3 helps us in finding the maximum no. of replicas that can be created for a task. We can be sure that at a time t, before the completion of the m^{th} last task, this observation holds true. Now for proving that this observation holds good for subsequent times as well, we use the tool the tool of mathematical induction.

According to the principle of mathematical induction, If f(1) is true, f(n+1) is true if f(n) is true, then f is true for all N.

Here since we know that the observation holds good at one particular time instant t, we'll assume that the result is true for a time instant t' and prove that the observation is true for a later instant t''.

At t', we can expect 2 cases,

i. All replica groups are of the same size s.

ii. Some replica groups are of size s, and some s+1.

Let t'' be the earliest completion time of a task after t'. Let 'u' be the completed task. Let I (respectively J) be the set of the replica groups at time t' with size s (respectively s+1) except the instance group of 'u'. Then at time t'', the algorithm terminates all the remaining task instances of 'u' and increases the number of replicas of the tasks which are not completed until time t''. Regardless of the size of the instance group of 'u', first of all, the algorithm one by one increases replicas of the tasks of which the instance group is in I. If free processors remain after the size of every instance group in I is increased by one, then the algorithm one by one increases replicas of the tasks of which the instance group is in J. If free processors still remain after the size of every instance group in J is increased by one, then this algorithm one by one increases replicas of the tasks of which the instance group is in I. The algorithm repeats the above increment process until free processors run out.

Hence we find that the observation is true for time t'' when we assume it is true for t'. Hence by Induction, the observation holds good at all times. During the execution of the xth last task, m task instances of exactly x tasks are being executed. These m task instances include exactly x originals. From the above statements we can judge that the number of replicas of a task (excluding the originals) is given by (m-x)/x. From observation 3, we shall find that this value can be exceeded by 1 at the maximum. Hence, the total number of replicas of the xth last task is given as

Floor ((m-x)/x) <= Replicas <= ceil ((m-x)/x)

where the floor statement takes care of rounding decimals to its nearest lower integer and the ceil, takes into account the criterion specified by observation 3. So, total number of replicas from the observation 2 can be given by

r = Σ replicas (summation of last tasks from 1 to m-1)
<=Σ ceil((m-x)/x) (summation of last tasks from 1 to m-1)
< Σ ((m-x)/x + 1) (summation of last tasks from 1 to m-1)
=Σ (m/x) (summation of last tasks from 1 to m-1)

VI. ANALYSIS

The TPCC of a schedule generated by our algorithm is atmost (1+ (m ln(m-1)+m)/n) times the optimal TPCC.

6.1 Analysis Of TPCC Ratio Using Matlab

• If m is fixed, then the approximation ratio of RR decreases suddenly with an increase in n.
• If n is fixed, then the approximation ration of RR increases gradually with an increase in m. The above analysis can be understood from the Table 2.

Figure 4 shows the variation of TPCC ratio with 4 processors and variable number of tasks.

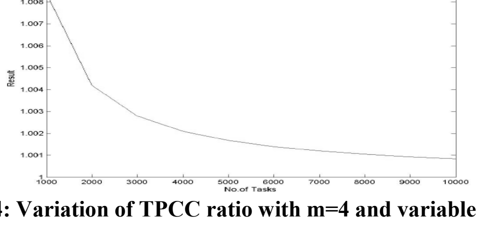

Fig. 4: Variation of TPCC ratio with m=4 and variable n.

No.of proc.	4	8	16	32
No.of tasks				
1000	1.008394	1.023567	1.059329	1.141888
2000	1.004197	1.011784	1.029664	1.070944
3000	1.002798	1.007856	1.019776	1.047296
4000	1.002099	1.005892	1.014832	1.035472
5000	1.001679	1.004713	1.011866	1.028378
6000	1.001399	1.003928	1.009888	1.023648
7000	1.001199	1.003367	1.008476	1.020270
8000	1.001049	1.002946	1.007416	1.017736
9000	1.000933	1.002619	1.006592	1.015765
10000	1.000839	1.002357	1.005933	1.014189

Table 2: Processors and tasks

6.2 Simulation Results:

Input Considered:

Enter the No.of tasks 7
Enter the No.of Processors 4
Enter the processor weightage 0.4
Enter the processor weightage 0.8
Enter the processor weightage 0.1
Enter the processor weightage 0.5
Enter the length of the task 20
Enter the length of the task 20
Enter the length of the task 20
Enter the length of the task 20
Enter the length of the task 20
Enter the length of the task 20
Enter the length of the task 20

Observations from the simulation:

i. The task replica algorithm suggested by us utilizes the available processor cycle at any instant of time and the processors are not idle till the schedule gets completed.

ii. The task utilization graph of the task replica algorithm follows the TPCPS (Total processor cycle consumption per second) till the schedule gets completed.

Makespan of task replica algorithm is less than the primitive algorithm. Figure 5 shows the task utilizations in primitive and proposed algorithms.

Output Obtained:
Makespan in Primitive algorithm = 32
TPCC in Primitive algorithm = 318

VII. CONCLUSION

The obtained TPCC value is found to be slightly greater that the ideal value nL. But the proposed algorithm reduces both TPCC and makespan. Our future works include incorporating the changes necessary for resolution of task dependency problem in real time systems.

t	Processors	Task Length in PA	Task length in OA
0	3 1 1 1	17 19 19 19	17 19 19 19
1	2 3 1 1	15 16 18 18	15 16 18 18
2	3 8 1 4	12 8 17 14	12 8 17 14
3	2 4 0 4	10 4 17 10	10 4 17 10
Task(OA) 5 allocated to processor no. 2			
Task(PA) 5 allocated to processor no. 2			
4	2 8 1 3	8 16 16 7	8 16 16 7
5	3 2 1 5	5 14 15 2	5 14 15 2
Task(OA) 6 allocated to processor no. 4			
Task(PA) 6 allocated to processor no. 4			
6	3 2 1 4	2 12 14 18	2 12 14 18
7	1 2 1 4	1 10 13 14	1 10 13 14
Task(OA) 7 allocated to processor no. 1			
Task(PA) 7 allocated to processor no. 1			
8	2 5 1 1	19 5 12 13	19 5 12 13
9	0 3 0 2	19 2 12 11	19 2 12 11
10	3 1 0 4	16 1 12 7	16 1 12 7
Task(OA) 3 is replicated to processor no. 2			
11	3 1 0 1	13 0 12 6	13 20 12 6
12	2 1 0 4	11 0 12 2	11 19 12 2
Task(OA) 3 is replicated to processor no. 4			
13	3 6 0 3	8 0 12 0	8 13 12 19
14	2 3 0 4	6 0 12 0	6 10 12 15
15	3 6 1 3	3 0 11 0	3 4 11 12
Task(OA) 7 is replicated to processor no. 2			
Task(OA) 7 is replicated to processor no. 3			
Task(OA) 7 is replicated to processor no. 4			
16	1 4 1 5	2 0 10 0	2 20 20 19
17	1 2 1 2	1 0 9 0	1 18 19 17
18	4 8 0 2	0 0 9 0	0 0 0 0
All Tasks Completed execution			
Makespan in Proposed Algorithm = 17.250000			
TPCC in Proposed Algorithm = 167.500000			
18	4 8 0 2	0 0 9 0	0 0 0 0
19	0 1 1 1	0 0 8 0	0 0 0 0
20	4 2 0 5	0 0 8 0	0 0 0 0
21	2 3 0 2	0 0 8 0	0 0 0 0
22	2 5 0 1	0 0 8 0	0 0 0 0
23	2 7 1 2	0 0 7 0	0 0 0 0
24	1 7 0 4	0 0 7 0	0 0 0 0
25	4 6 1 1	0 0 6 0	0 0 0 0
26	1 7 1 5	0 0 5 0	0 0 0 0
27	3 6 0 1	0 0 5 0	0 0 0 0
28	1 2 1 5	0 0 4 0	0 0 0 0
29	2 7 1 3	0 0 3 0	0 0 0 0
30	2 4 1 1	0 0 2 0	0 0 0 0
31	1 1 1 4	0 0 1 0	0 0 0 0

Table 3: Output Table

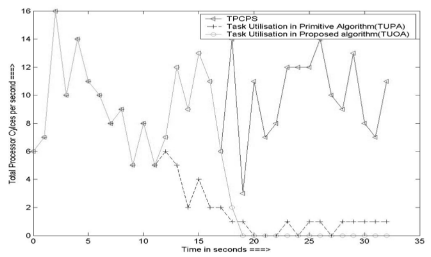

Fig. 5: Comparison of task utilizations in primitive and proposed algorithm

VIII. REFERENCE

[1] *Scheduling Algorithms for Grid Computing: State of the Art and Open Problems*. Fangpeng Dong and Selim G. Akl School of Computing

[2] Queen's University Kingston, Ontario January 2006. Technical Report No. 2006-504

[3] R. Buyya and D. Abramson and J. Giddy and H. Stockinger, *Economic Models for Resource Management and Scheduling in Grid Computing*, in J. of Concurrency and Computation: Practice and Experience, Volume 14, Issue.13-15, pp. 1507-1542, Wiley Press, December 2002.

[4] H. El-Rewini, T. Lewis, and H..Ali, *Task Scheduling in Parallel and Distributed Systems*, ISBN: 0130992356, PTR Prentice Hall, 1994.

[5] D.P. Spooner, J. Cao, J.D. Turner, H. N. L. C. Keung, S.A. Jarvis and G.R. Nudd, *Localised Workload Management using Performance*, in the Proc. of the 18th Annual UK Engineering Workshop (UKPEW' 2002), University of Glasgow, UK, July 2002.

[6] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Saho, S. Smallen, N. Spring, A. Su and D. Zadorodnov, " Adaptive Computing on the Grid Using Apples" in *IEEE Trans. On Parallel and Distributed Systems (TPDS),* Vol 14, No.4, pp. 369-382, 2003