Naveen Kumar Laskari, Aparna Tanam, Ranganath K / International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622 www.ijera.com
Vol. 1, Issue 4, pp. 1536-1546

Hierarchical Scheduling for Multiprocessor Systems

Naveen Kumar Laskari*, Aparna Tanam**, Ranganath K***

*(Department of CSE, HITAM, Hyderabad, A.P
**(Department of CSE, JBIET, Hyderabad, A.P
***(Department of CSE, HITAM, Hyderabad, A.P

ABSTRACT

Parallel and distributed computing [multi-chip multiprocessor] environments are essential and utilized to meet the needs of a wide variety of high-throughput applications. Scheduling strategies are important in order to efficiently utilize the resources and to improve response times, throughput and utilization of computing platforms. In this paper, we present a two-level hierarchical method for scheduling of independent coarse-grained tasks in multi-chip multiprocessor environments. A set of jobs that is feasible on some uniform multiprocessor platform with cumulative computing capacity and in which the fastest processor has speed s < 1 is schedulable on an SMP composed by m processors with unit capacity assuming an arbitrary collection of jobs. A task system composed by periodic and sporadic tasks with constrained deadlines is feasible on a uniform multiprocessor platform that has S ¼ total and s ¼ max. With two-level architecture, the scheduler (master node of upper-level) proceeds with distribution of tasks to computing sites. While the Local Resource Manager (master node of lower level) assigns this task to an available computing node according to a given threshold. Comparing experimental results with those obtained from well known traditional scheduling algorithms, the effectiveness of the proposed method consistently shows a benefit from this approach.

Keywords- **Cluster, DAG, Grid, Multiprocessor, Scheduling, Network of Workstations (NOW)**

I. INTRODUCTION

Heterogeneous computing changes a network of heterogeneous computers into a single computing resource entity. The central theme of heterogeneous computing is to utilize computing resources of different machine architectures. On one hand, many users find that the computers they use are not powerful enough to meet their purposes; on the other hand, many of the computers in a typical network are idle, having no job to process. This situation happens within a LAN, or even WAN-wide. Ideally, if computing resources can be shared, it could dramatically increase our work efficiency. The greatest challenge to network computing is to obtain a near-optimal algorithm to solve the mapping and scheduling problem. Several characteristics should be considered i.e. the dynamic nature of computer traffic loading, the intensity of task submissions, the infrastructure of the computer network and the fair competition for utilizing computational resources.

II. LITERATURE SURVEY

2.1 HETEROGENEOUS NETWORK ENVIRONMENTS

Recent advances in software and hardware technology have greatly improved the performance of a Network of Workstations (NOW). Very often in a NOW environment, machines are owned by individual users whose typical processing needs rarely require the full capacity of their workstation. Conversely, some users may have computationally intensive tasks that are beyond the capacity of the workstation he or she owns. Consequently, if each user were restricted to run tasks within the boundaries of a single workstation, precious computational resources would be wasted. This raises the challenge of developing a load-balancing environment to utilize the available computational resource more efficiently.

The nature of a connected workstation network is heterogeneous. Heterogeneity takes a number of forms:

1) Heterogeneity of a configuration, whereby hosts may have different processing power, memory space, disk storage, and so on.

2) Architectural heterogeneity that makes it impossible to execute the same code on different hosts.

3) Operating system heterogeneity, where hosts have different operating systems running and may be incompatible.

However, for this paper, only the heterogeneity of a configuration was considered, in which we assume that a task can be executed on any computer node in the NOW. Besides heterogeneity, a NOW system has three other unique features in comparison with a multiprocessor or a multi-computer system:

1) Low bandwidth communication: Even when high-speed networks are used, the inter-node communication still causes bottleneck problems. Therefore, only coarse-grained or medium-grained parallel tasks are suitable for running on a NOW.

2) Random network topology: A NOW system connects workstations in a random way and its topology may change from time to time in practice.

3) Multidirectional scaling: A NOW system can be scaled in three directions: by increasing the number of workstations, by upgrading the power of the workstations, and by a combination of both.

2.2 DAG MODEL

In this paper, we define a task as an independent, computationally intensive application sent by different users. A parallel task can be divided into subtasks with data dependence between them. By the loop-unraveling technique, computational loops can be subdivided into a number of subtasks. Usually a large class of data-flow computation problems and many numerical algorithms (such as matrix multiplication) do not have conditional branches or indeterminism in the program, thereby making them suitable candidates for subdivision. In addition, in many numerical tasks, such as Gaussian elimination or fast Fourier transforms (FFT), the loop bounds are known during compile-time. As such, one or more iterations of a loop can be deterministically encapsulated in a subtask. These techniques make parallel processing of a task possible. Based on the discussion above, a parallel task can be represented by a Directed Acyclic Graph (DAG), which is illustrated in Figure 1. In a DAG, V is a set of v nodes and E is a set of e directed edges. G= (V, E), where the set of vertices V= {v1, v2, vn} represents the set of subtasks to be executed, and the set of weighted, directed edges E represents communication between subtasks. A node in the DAG represents a subtask that is a set of instructions that must be executed sequentially without preemption in the same processor. The weight of a node is computation cost. The edges in the DAG correspond to the communication messages and precedence constraints among the nodes. The weight of an edge is referred as communication cost. Thus indicates communication from subtask vi to vj, and |eij| represents the volume of data sent between these subtasks. The node and edge weights are usually obtained by estimation using profiling information of operations such as numerical operations, memory access operations, and message-passing primitives. In a DAG, the source node of an edge is called the parent node while the sink node is called the child node. A node with no parent is called an entry node and a node with no child is called an exit node. As shown in Figure 1, N2 is the parent of N4 and N5, N4 and N5 are the child nodes of N2. N1 is the entry node, and N8 and N9 are exit nodes, and the line in bold is the crucial path of the task.

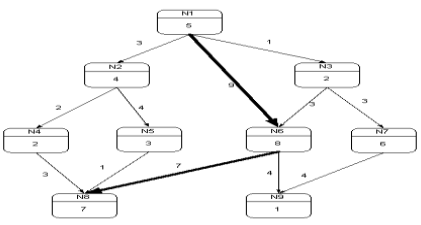


Figure 1: Task DAG Graph

Subtask processing can either be preemptive or non-preemptive. After a node has been selected for execution, non-preemptive subtask processing dictates that the subtask cannot be moved even if a more suitable node is available. In contrast, preemptive processing entails stopping the process, moving the subtask to the new node, and resuming its execution. Preemptive processing is much more costly than a non- preemptive transfer in two cases: First, the implementation and maintenance of the mechanisms necessary to encapsulate, transfer, and resume execution from this complex state are expensive. Second, since preemptive processing causes an overhead that is much greater than that of the non-preemptive variety, it is not obvious what performance improvement might result beyond non-preemptive processing. For this paper, a non-preemptive DAG represents a subtask structure that assumes that once a subtask starts on a machine, it cannot be stopped. If it is stopped for some unexpected reason, like machine failure, it has to be restarted again.

2.3 MAPPING AND SCHEDULING ALGORITHM

The problem of mapping and scheduling multiple tasks can be divided into two categories: task mapping and scheduling, and subtask mapping and scheduling. In task mapping and scheduling, independent tasks are scheduled among the network of workstations to optimize overall system performance. In contrast, the subtask scheduling and mapping problem requires the allocation of multiple interacting subtasks of a single parallel task in order to minimize the completion time. Task scheduling usually requires dynamic run-time scheduling because it is not a priori decidable, the subtask mapping and scheduling problem can be addressed both statically and dynamically. In this paper, a multiple task computing simulation in a heterogeneous environment is used; therefore both task and subtask mapping and scheduling are addressed.

2.3.1 TASK LEVEL MAPPING AND SCHEDULING

Task level mapping and scheduling considers a scenario where each task is independent, and there is no communication between them. Those independent tasks compete for computational resources, and the task level mapping and scheduling heuristics attempt to match these tasks with available computational entities. The task mapping heuristics can be grouped into two categories: dynamic heuristics and static heuristics. Dynamic heuristics can be further grouped into two categories: immediate mode and batch mode heuristics. In the immediate mode, a task is mapped as soon as it arrives. In the batch mode, tasks are not mapped as they arrive; instead they are collected into a set that is examined for mapping at prescheduled times called mapping events. The independent set of tasks that are considered for mapping at the mapping events is called a meta-task. A meta-task can include newly arrived tasks (i.e., the ones arriving after the last mapping event) and the ones that were mapped in earlier mapping events but did not begin execution. While immediate mode heuristics consider a task for mapping only once, batch mode heuristics consider a task for mapping at each mapping event until the task begins

execution. For immediate mode there is no mapping delay between mapping events, the tasks are mapped right after they arrive. However, as a tradeoff, since immediate mode can only map tasks once, its performance is not as good as batch mode when the arrival of tasks is very intensive. There are five different types of immediate mode heuristics. They are 1) minimum completion time (MCT); 2) minimum execution time (MET); 3) switching algorithm (SA); 4) k-percent best (KPB); and 5) opportunistic load balancing (OLB). The MCT heuristic assigns each task to the machine that results in that task's earliest completion time in order to balance the load. The MET heuristic assigns each task to the machine that performs that task's computation in the least amount of execution time. The MET heuristic can potentially create load imbalance across machines by assigning many more tasks to some machines than to others. The SA heuristic is a combination of MCT and MET. The idea behind it is that when the tasks are arriving in a random mix, it is possible to use the MET, at the expense of load balancing until a given threshold, and then use the MCT to smooth the load across the machines. SA uses the MCT and MET heuristics in a cyclical fashion depending on the load distribution across the machines. The purpose is to have a heuristic with the desirable properties of both the MCT and the MET. The KPB heuristic is another form of a combination of MET and MCT. The heuristic considers only a subset of machines while mapping a task. The subset is formed by picking the m (k/100) best machines based on the execution times for the task, where $100/m \leq k \leq 100$. The task is assigned to a machine that provides the earliest completion time in the subset. If k=100, then the KPB heuristic is reduced to the MCT heuristic. If k=100/m, then the KPB heuristic is reduced to the MET heuristic. The OLB heuristic is very simple; it assigns a task to the machine that becomes ready next, without considering the execution time of the task onto that machine. If multiple machines become ready at the same time, then one machine is arbitrarily chosen. Three batch mode heuristics are presented here: (i) the Min-min heuristic, (ii) the Max- min heuristic, and (iii) the Sufferage heuristic. The Min-min heuristic is archived by executing following step:

1) For each task find the earliest completion time and the machine that obtains it.

2) Within these earliest completion times, find the minimum, map the task to the machine.

3) Update computational entity free time.

4) Repeat step 1, 2, and 3 until all tasks are mapped.

The Max-min heuristic is similar to the Min-min heuristic. It differs from the Min-min heuristic in step 2, which instead of finding the minimum the Max-min heuristic is to find the maximum. The Max-min is likely to do better than the Min-min heuristic in cases where there are many shorter tasks than longer tasks. The Sufferage heuristic is based on the idea that better mappings can be generated by assigning a machine to a task that would "suffer" most in terms of expected completion time, if that particular machine is not assigned to it.

In contrast to dynamic task mapping heuristics, static heuristics perform task mapping statically (i.e., off-line, or a predictive manner). Static heuristics assume all tasks are known before they are mapped. The static OLB (opportunistic load balancing) heuristic is similar to its dynamic counterpart except that it assigns tasks in an arbitrary order, instead of order of arrival. The UDA (user directed assignment) heuristic works in the same way as the MET heuristic except that it maps tasks in an arbitrary order instead of order of arrival. The fast greedy heuristic is the same as the MCT, except that it maps tasks in an arbitrary order instead of their order of arrival. The static Min-min heuristic works in the same way as the dynamic Min-min, except a meta-task contains all the tasks in the system. The static Max-min heuristic works in the same way as the dynamic Max- min, except a meta-task has all the tasks in the system. The greedy heuristic performs both the static Min-min and static Max-min heuristics, and uses the better of the two solutions.

2.3.2. Subtask Level Mapping and Scheduling

Subtask level mapping and scheduling, also referred as DAG mapping and scheduling, considers a scenario where each subtask is related, and there is data dependence between them. These related subtasks compete for computational resources, and the subtask level mapping and scheduling heuristics are to match these tasks with available computational entities and increase overall system performance and computational usage. In DAG scheduling, the target system is assumed to be a network of workstations, each of which is composed of a processor and a local memory unit; they do not share memory and communication between them relies solely on message-passing. The processors may be heterogeneous or homogeneous. However, DAG scheduling assumes every module of a parallel program can be executed on any workstation even though the completion times on different processors may be different. The workstations are connected by an interconnection network with a certain topology. The topology may be fully-connected or of a particular structure such as a hypercube or mesh [Y. Kwok 99]. Subtask mapping and scheduling algorithms exist in two forms: static and dynamic. As mentioned, a parallel task can be represented by a DAG. In static scheduling, which is usually done at compile time, the characteristics of a task are known before program execution. In dynamic scheduling, a few assumptions about the task can be made before execution. Dynamic schedulers usually offer better performance, but the goal of a scheduling algorithm includes not only the minimization of the program completion time but also the minimization of the scheduling overhead. Most scheduling algorithms are based on the list scheduling techniques. The basic idea of list scheduling is to make a scheduling list (a sequence of subtasks for scheduling) by assigning them some priorities, and then schedule those subtasks according to their priorities. The two frequently used attributes for assigning priority are the t-level (top level), b-level (bottom level), and p-level (partial level). The t-level of a node is the length of a longest path from an entry node to the node itself.

Here, the length of a path is the sum of all the node and edge weights along the path. The b-level of a node is the length of the longest path (there can be more than one longest path) to an exit node. Some scheduling algorithms do not take into account the edge weights in computing the b-level. In such a case, the b-level does not change throughout the scheduling process. This algorithm is referred to as the static b- level. The p-level of a node is simply the computation cost of that given node; also, the p-level does not change throughout the scheduling process, as it is illustrated in Figure 1.

Node	t-level	b-level	p-level
N 1	0	36	5
N 2	8	19	4
N 3	6	18	2
N 4	14	12	2
N 5	16	11	3
N 6	14	22	8
N 7	11	11	6
N 8	26	7	7
N 9	29	1	1

Table 1. T-levels, b-levels, and p-levels for the DAG of Figure 1

Different algorithms use the t-level and b-level in different ways. Some algorithms assign a higher priority to a node with a smaller t-level while some algorithms assign a higher priority to a node with a larger b-level, or a larger p-level. Still some algorithms assign a higher priority to a node with a larger (b-level – t-level). In general, scheduling in a descending order of b-level tends to schedule critical path nodes first, while scheduling in an ascending order of t-level tends to schedule nodes in a topological order. The composite attribute (b-level – t-level) is a compromise between the previous two cases. The notion behind the p-level was that by executing higher computationally intensive subtasks first, the overall completion time of the task may be minimized. List scheduling includes both static list scheduling and dynamic list scheduling. In static list scheduling, the scheduling list is statically constructed before node allocation begins, and most importantly, the sequencing in the list is not modified. A task is usually scheduled on the processor that gives the earliest start time for the given task. Thus, at each scheduling step, the task is selected first, then its destination processor. The procedure of static list scheduling entails repeatedly executing the following two steps until all the nodes in the graph are scheduled: 1) removing the first node from the scheduling list; 2) allocating the node to a processor which allows the earliest start-time. Dynamic list scheduling takes a different approach. After each allocation, the priorities of all unscheduled nodes are re-computed, and consequently the scheduling list is then rearranged. In this case, the tasks do not have a pre-computed priority. At each scheduling step, each ready task is tentatively scheduled to each processor, and the best task-processor pair is selected. Both the task and its destination processor are selected at the same time. Thus, these algorithms essentially employ the following three-step approaches: 1) determining new priorities of all unscheduled nodes; 2) selecting the node with the highest priority for scheduling; 3) allocating the node to the processor that allows the earliest start-time or earliest finish-time. Scheduling algorithms that employ this three-step approach can potentially generate better schedules, but the tradeoff is the scheduling time is increased.

Both static and dynamic approaches of list scheduling have their advantages and drawbacks in terms of the schedule quality they produce. Static approaches are better suited for communication-intensive and irregular problems, where selecting important tasks first is more crucial. Dynamic approaches are better suited for computationally intensive applications with a high degree of parallelism, because these algorithms focus on obtaining good processor utilization.

2.3.3 MULTIPLE TASK MAPPING AND SCHEDULING

In this paper, we analyze the behavior of multiple task (multiple DAG) computing in a heterogeneous environment, therefore the objective of this research is to study multiple DAG scheduling. However, there is little literature in this area. Iverson presents a dynamic, competitive scheduling of multiple DAGs [IvÖ98]. In his framework, each task is responsible for scheduling (0f)its own tasks. Thus, there is no centralized scheduling authority. A task is scheduled without the knowledge of other tasks; the task scheduler only knows the current workload of the network. Iverson's algorithm is based on the expectation that if each task had the best mapping and scheduling possible, the overall parallel computing performance would be optimal.

III. PROBLEM STATEMENT

Aiming to provide a non-preemptive scheduling to minimize the maximum completion time given a set of independent computational tasks, to obtain acceptable performance and allocation of application processes to the processors available

3.1 EXISTING SYSTEM

A Symmetric Multiprocessor (SMP) system consisting of m processors is addressed. The problem of preemptively scheduling a real-time task set on these systems can be solved in two different ways: by partitioning tasks to processors or with a global scheduler. In the first case, tasks are allocated to processors at design time with an offline procedure. The partitioning problem is analogous to the bin packing problem, which is known to be NP-hard in the strong sense. However, once the tasks are allocated, the scheduling problem is reduced to m single-processor scheduling problems, for which optimal solutions are known when preemptions are allowed. The main advantage of this approach is, its simplicity and efficiency. The efficiency of

the system depends on the frequency at which load-balancing routines are called and on the complexity of these algorithms. An alternate solution is a global work-conserving scheduler where migration from one processor to another is allowed during a task lifetime.

3.2 LIMITS OF THE EXISTING SYSTEM

Global Scheduling algorithms are based on the concept of quantum (or slot) and at each quantum, the scheduler allocates tasks to processors. A disadvantage of this approach is that all processors need to synchronize at the quantum boundary, when the scheduling decision is taken. Moreover, if the quantum is small, the overhead in terms of the number of context switches and migrations may be too high. To obtain inexpensive computational cycles, grid technology has emerged to fulfill the needs for solving large-scale computing intensive high-throughput applications[1], through the aggregation of a number of available resources. Task independent applications such as data mining, Monte Carlo, image manipulations are most suitable class of applications that uses a wide spectrum of techniques like branch & bound, integer programming, searching, graph theory& randomization.

3.3 PROPOSED SOLUTION

A method is developed, aiming to provide a non-preemptive scheduling to minimize the maximum completion time. We perform the experiment using a widely used scheduling simulator, then present and compare our proposed algorithm with two traditional well-known scheduling algorithms. The proposed algorithm generates in most of cases better solutions than the referenced algorithms in terms of the maximum completion times. We have developed a method for scheduling of task independent parallel applications in Multiprocessor Environments. Essentially, the Task Threshold-based Mapping method (TMM), tasks are distributed to computing nodes based on defined thresholds.

3.3.1 BASIC CONCEPTS AND TERMINOLOGY

Although many types of resources can be shared and used in a Computational Multiprocessor Model, normally they are accessed through an application running in the grid. Normally, an application is used to define the piece of work of higher level in the Grid. A typical grid scenario is as follows: an application can generate several jobs, which in turn can be composed of sub-tasks, in order to be solved; the Multiprocessor System is responsible for sending each sub-task to a resource to be solved. In a simpler grid scenario, it is the user who selects the most adequate machine to execute its sub-tasks. However, in general, Multiprocessor Systems must dispose of schedulers that automatically and efficiently find the most appropriate machines to execute an assembly of tasks.

3.3.2 SCHEDULING PROBLEMS IN COMPUTATIONAL MULTIPROCESSOR MODELS

Rather than a problem, scheduling in Multiprocessor Systems can be viewed as a whole family of problems. This is due to the many parameters that intervene scheduling as well as to the different needs of Grid-enabled applications. In the following, we give some basic concepts of scheduling in Multiprocessor Systems and identify most common scheduling types. Needless to say, job scheduling in its different forms is computationally hard; it has been shown that the problem of finding optimum scheduling in heterogeneous systems is in general NP-hard.

3.3.3 NEW CHARACTERISTICS OF SCHEDULING IN GRIDS

The scheduling problem in distributed systems is not new at all; as a matter of fact it is one of the most studied problems in the optimization research com- munity. However, in the grid setting there are several characteristics that make the problem different from its traditional version of conventional distributed systems. Some of these characteristics are the following:

• The dynamic structure of the Computational Multiprocessor Model. Unlike traditional distributed systems such as clusters, resources in a Multiprocessor System can join or leave the Grid in an unpredictable way. It could be simply due to loosing connection to the system or because their owners switch off the machine or change the operating system, etc. Given that the resources cross different administrative domains, there is no control over the resources.

• The high heterogeneity of resources. Multiprocessor Systems act as large virtual super- computers, yet the computational resources could be very disparate, ranging from laptops, desktops, clusters, supercomputers and even small devices of limited computational resources. Current Grid infrastructures are not yet much versatile but heterogeneity is among most important features to take into account in any Multiprocessor System.

• The high heterogeneity of jobs. Jobs arriving to any Multiprocessor System are diverse and heterogeneous in terms of their computational needs. For instance, they could be computing intensive or could be data intensive; some jobs could be full applications having a whole range of specifications other could be just atomic tasks. Importantly, Multiprocessor System could not be aware of the type of tasks, jobs or applications arriving in the system.

• The high heterogeneity of interconnection networks. Grid resources will be connected through Internet using different interconnection networks. Trans- mission costs will often be very important in the overall Grid performance and hence smart ways to cope with the heterogeneity of interconnection networks is necessary.

• The existence of local schedulers in different organizations or resources. Grids are expected to be constructed by the "contribution" of computational resources across institutions, universities, enterprises and individuals. Most of these resources could eventually be running local applications and use their local schedulers, say, a Condor batch system. In

such cases, one possible requirement could be to use the local scheduler of the domain rather than an external one.

• The existence of local policies on resources. Again, due to the different owner- ship of the resources, one cannot assume full control over the Grid resources.

Companies might have unexpected computational needs and may decide to reduce their contribution to the Grid. Other policies such as rights access, available storage, pay-per-use, etc. are also to be taken into account.

• Job-resource requirements: Current Grid schedulers assume full availability and compatibility of resources when scheduling. In real situations, however, many restrictions and/or incompatibilities could be derived from job and resource specifications.

• Large scale of the Multiprocessor System: Multiprocessor Systems are expected to be large scale, joining hundreds or thousands of computational nodes world-wide. Moreover, the jobs, tasks or applications submitted to the Grid could be large in number since different independent users and/or applications will send their jobs to the Grid without knowing previous workload of the system. Therefore, the efficient management of resources and planning of jobs will require the use of different types of scheduling (super-schedulers, meta-schedulers, decentralized schedulers, local schedulers, resource brokers, etc.) and their possible hierarchical combinations.

• Security: This characteristic, which exists in classical scheduling, is an important issue in Multiprocessor Scheduling. Here the security can be seen as a two-fold objective: on the one hand, a task, job or application could have a security requirement to be allocated in a secure node, that is, the node will not "watch" or access the processing and data used by the task, job or application. On the other hand, the node could have a security requirement, that is, the task, job or application running in the resource will not "watch" or access other data in the node.

3.4 PHASES OF SCHEDULING IN GRIDS

In order to perform the scheduling process, the Grid scheduler has to follow a series of steps which could be classified into five blocks: (1) Preparation and information gathering on tasks, jobs or applications submitted to the Grid; (2) Resource selection; (3) Computation of the planning of tasks (jobs or applications) to selected resources; (4) Task (job or application) allocation according to the planning (the mapping of tasks, jobs or applications to selected resources); and, (5) Monitoring of task, job or application completion (the user is referred to for a detailed description). The Grid scheduler will have access to the Multiprocessor Information on available resources and tasks, jobs or applications (usually known as "Multiprocessor Information Service" in the Grid literature). Moreover, the scheduler will be informed about updated information (according to the scheduling mode). This information is crucial for the scheduler in order to compute the planning of tasks, jobs or applications to the resources. Resource selection: Not all resources could be candidates for allocation of task, jobs or applications. Therefore, the selection process is carried out based on job requirements and resource characteristics. The selection process, again, will depend on the scheduling mode. For instance, if tasks were to be allocated in a batch mode, a pool of as many as possible candidate resources will be identified out of the set of all available resources. The selected resources are then used to compute the mapping that meets the optimization criteria. As part of resource selection, there is also the advanced reservation of resources. Information about future execution of tasks is crucial in this case. Although the queue status could be useful in this case, it is not accurate, especially if priority is one of the task requirements. Another alternative is using prediction methods based on historical data or user's specifications of job requirements.

3.4.1 TASK ALLOCATION

In this phase the planning is made effective: tasks (jobs or applications) are allocated to the selected resources according to the planning.

3.4.2 TASK EXECUTION MONITORING

Once the allocation is done, the monitoring will inform about the execution progress as well as possible failures of jobs, which depends on the scheduling policy will be rescheduled again (or migrated to another resource).

3.5 COMPUTATION MODELS FOR FORMALIZING MULTIPROCESSOR SCHEDULING

Given the versatility of scheduling in Multiprocessor environments, one needs to consider different computation models for Multiprocessor Scheduling that would allow to formalize, implement and evaluate either in real Grid or through simulation, different scheduling algorithms. We present some important computation models for Multiprocessor Scheduling. It should be noted that such models have much in common with computation models for scheduling in distributed computing environments. We notice that in all the models described below, tasks, jobs or applications are submitted for completion to a single resource.

3.5.1 EXPECTED TIME TO COMPUTATIONAL MODEL

In the model proposed by Ali et al. [5], it is assumed that we dispose of estimation or prediction of the computational load of each task (e.g. in millions of instructions), the computing capacity of each resource (e.g. in millions of instructions per second, MIPS), and an estimation of the prior load of each one of the resources. Moreover, the Expected Time to Compute matrix ETC of size number of tasks by number of machines, where each position ET C [t][m] indicates the expected time to compute task t in resource m, is assumed to be known or computable in this model. In the simplest of cases, the entries ETC [t][m] could be computed by dividing the workload of task t by the computing capacity of resource m. This formulation is usually feasible, since it is possible to know the computing capacity of resources while the computation need of the tasks (task workload) can be known

from specifications provided by the user, from historic data or from predictions

3.5.2 Modelling heterogeneity and consistency of computing

The ETC matrix model is able to describe different degrees of heterogeneity in distributed computing environment through consistency of computing. The consistency of computing refers to the coherence among execution times obtained by a machine with those obtained by the rest of machines for a set of tasks. This feature is particularly interesting for Multiprocessor Systems whose objective is to join in a single large virtual computer different resources ranging from laptops and PCs to clusters and supercomputers. Thus, three types of consistency of computing environment can be defined using the properties of the ETC matrix: consistent, inconsistent and semi-consistent. An ETC matrix is said to be consistent, if for every pair of machines mi and mj , if mi executes a job faster than mj then mi executes all the jobs faster than mj. In contrast an inconsistent ETC matrix, a machine mi may execute some jobs faster than another machine mj and some jobs slower than the same machine mj . Partially-consistent ETC matrices are inconsistent matrices having a consistent sub-matrix of a predefined size. Further, the ETC matrices are classified according to the degree of job heterogeneity, machine heterogeneity and consistency of computing. Job heterogeneity expresses the degree of variance of execution times for all jobs in a given machine. Machine heterogeneity indicates the variance of the execution times of all machines for a given job.

3.5.3 Multiprocessor Information System model

The computation models for Multiprocessor Scheduling presented so far allow for a precise description of problem instance however they are based on predictions, distributions or simulations. Currently, other Multiprocessor Scheduling models are developed from a higher level perspective. In the Multiprocessor Information System model the Grid scheduler uses task (job or application file descriptions) and resource file descriptions as well as state information of resources (CPU usage, number of running jobs per grid resource), provided by the Multiprocessor Information System. The Grid scheduler then computes the best matching of tasks to resources based on the up-to-date workload information of resources. This model is more realistic for Multiprocessor Environments and is especially suited for the implementation of simple heuristics such as FCFS (First Come First Served), EDF (Earliest Deadline First), SJF (Shortest Job First), etc.

3.5.4 Cluster and Multi-Cluster Grids model

Cluster and Multi-Cluster Grids refer to Grid model in which the system is made up of several clusters. For instance the Cluster Grid of an enterprise comprises different clusters located at different departments of the enterprise. One main objective of cluster grids is to provide a common computing infrastructure at enterprise or department levels in which computing services are distributed to different clusters. More generally, clusters could belong to different enterprises and institutions, that is, are autonomous sites having their local users (both local and grid jobs are run on resources) and usage policies. The most common scheduling problem in these models is a Grid scheduler which makes use of local schedulers of the clusters. The benefit of cluster grids is to maximize the usage of resources and at the same time, increase of throughput for user tasks (jobs or applications).

3.6 Multiprocessor System Performance and Scheduling Optimization Criteria

Several performance requirements and optimization criteria can be considered for Multiprocessor Scheduling problem the problem is multi-objective in its general formulation. We could distinguish proper Multiprocessor System performance criteria from scheduling optimization criteria although both performance and optimization objectives allow to establish the overall Multiprocessor System performance. Multiprocessor System performance criteria include: CPU utilization of Grid resources, load balancing, system usage, queuing time, throughput, turnaround time, cumulative throughput (i.e. cumulative number of completed tasks) waiting time and response time. In fact other criteria could also be considered for characterizing Multiprocessor System's performance such as deadlines, missed deadlines, fairness, user priority, resource failure, etc. Scheduling optimization criteria include: makespan, flowtime, resource utilization, load balancing, matching proximity, turnaround time, total weighted completion time, lateness, weighted number of hardy jobs, weighted response time, etc. Both performance criteria and optimization criteria are desirable for any Multiprocessor System; however, their achievement also depends on the considered model (batch system, interactive system, etc.). Importantly, it should be stressed that these criteria are conflicting among them; for instance, minimizing makespan conflicts with resource usage and response time. Among most popular and extensively studied optimization criterion is the minimization of the makespan. Makespan is an indicator of the general productivity of the Multiprocessor System: small values of makespan mean that the scheduler is providing good and efficient planning of tasks to resources. Considering makespan as a stand-alone criterion necessarily may not imply optimization of other objectives. As mentioned above, its optimization could in fact go in detriment to other optimization criteria. Another important optimization criterion is that of flowtime, which refers to the response time to the user submissions of task executions. Minimizing the value of flowtime means reducing the average response time of the Multiprocessor System. Essentially, we want to maximize the productivity (throughput) of the grid and at the same time we want to obtain planning of tasks to resources that offer an acceptable QoS.

Makespan, completion time and flowtime

In Multiprocessor Scheduling we can minimize the makespan and flowtime. Makespan is the time when finishes the latest task and flowtime is the sum of finalization times of all the tasks. Formally they can define as:

minimization of makespan :

minSi ∈ Sched{maxj ∈ Jobs Fj }

Where Fj denotes the time when the task j finalizes, Sched is the set of all possible schedules and Jobs the set of all jobs to be scheduled. Note that makespan is not affected by any particular execution order of tasks in a concrete resource, while in order to minimize flowtime of a resource, tasks that have been assigned to should be executed in a ascending order of their workload (computation time).Completion time of a machine m is the time in which machine m will finalize the processing of the previous assigned tasks as well as of those already planned tasks for the machine. This parameter measures the previous workload of a machine. Notice that this definition requires knowing both the ready time for a machine and the expected time to complete of the tasks assigned to the machine. The expression of makespan, flowtime and completion time depends on the computational model. For instance, in the ETC model, completion[m] is calculated as follows: where ready times[m] is the time when machine m will have finished the previously assigned tasks.
Makespan can be expressed in terms of the completion time of a resource, as follows:

Makespan = Max {completion[i]/i€ Machines}

Similarly, for the flowtime we use the completion times of machines, but now by first sorting in ascending order according to their ETC values the tasks assigned to a machine.

3.7 Proposed Scheduling Model

As one of the means to obtain inexpensive computational cycles, grid technology has emerged to fulfill the needs for solving large-scale computing intensive high-throughput applications, through the aggregation of a number of available resources. Multiple users can simultaneously utilize any of resources interconnected to execute these large parallel applications. To effectively utilize hybrid heterogeneous computational resources, resource management and task scheduling are fundamental factors for achievements in grids. Due to wide distribution and heterogeneity characteristics of grid platforms, loosely coupled parallel applications are better suited for execution on this platform than tightly coupled ones [5, 6]. In particular, task independent applications such as data mining, Monte Carlo, image manipulation are most suitable class of applications for current design of Multiprocessor Environments. Scheduling task independent applications is still far to be considered well-established. Finding optimal scheduling is an NP-complete problem, and researchers have still resorted to devising efficient heuristics. A number of heuristics have been proposed based on a wide spectrum of techniques, including branch-and-bound, integer-programming, searching, graph-theory, randomization, genetic algorithms, and evolutionary methods [3, 4]. These algorithms are based on diverse assumptions; they differ in their functionalities as well. Simulation and modeling have been dedicated and extensively used by professionals in different application fields, particularly, in the area of computer science. e.g., for microprocessor design and network protocol design, in which simulation and modeling have been used for decades. They are convenient and cost effective. In Multiprocessor computing, several widely used and acknowledged simulations have been commonly used to evaluate tasks scheduling and load balancing. In this paper, a promising method is developed, aiming to provide a non-preemptive scheduling to minimize the maximum completion time (the schedule length or makespan) ,given a set of independent computational tasks to obtain acceptable performance is a good allocation of application processes to the processors available. We perform the experiment using a widely used scheduling simulator, then present and compare our proposed algorithm with two traditional well-known scheduling algorithms. The proposed algorithm generates in most of cases better solutions than the referenced algorithms in terms of the maximum completion times. Task scheduling in dynamic and heterogeneous computing environment such as Grid is not trivial, since major concerns that arise during the analysis and development of strategies for such purpose is to search alternatives to improve throughput and utilization in these computing platforms. Looking at the nature of task independent applications, the scheduling process may seem to be easy due to its simplicity. However, due to dynamic behavior and heterogeneity of resources, not only they may not provide similar performance for all applications, but also contention created among applications running on same shared resources, causing delays and affecting the quality of service [2] [6].

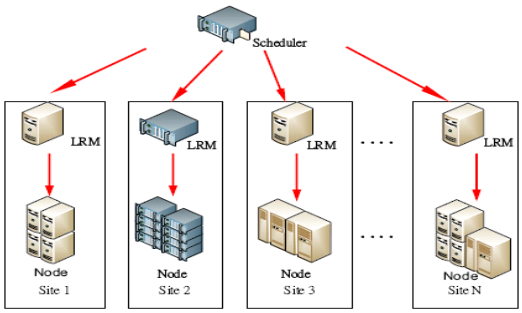


Figure 2: Proposed two level scheduling method.

IV. Implementation

Given a set of independent tasks, each task of this set is classified into one of five defined levels, say A, B, C, D and E (A is most time demanding while E is less demanding), according to the execution time needed. Computing nodes in each site of grid platform are rated according to their computing power, that is, given MCC as Maximum Computing Capability of any node in a grid platform, computing nodes are rated and classified in a particular class if this node's computing power is X% of MCC.

Node Computing Capability	Level of Classification
0%~20% MCC	E
21%~40% MCC	D
41%~60% MCC	C
61%~80% MCC	B
81%~100% MCC	A

Table 2- Level of Classification

In order to classify a computing site, we just need to look at highest rank achieved by any of computing nodes inside this site. For instance, a site contains 3 computing nodes, with levels of classification B, C, and D. Thus, this site is ranked with level B. That is, Level Classification SITE = max level {node1, node2… node n} our proposed method work as follows. A task is distributed, and shown next to Grid scheduler. Based on Round Robin technique, the Grid scheduler selects next suitable site to the execution of that given task, matching a suitable site to the demand need for the given task. As for matching process, task ranked with level A is expected to be distributed to a site ranked with level A, while task ranked with level D is expected to be distributed to sites ranked with levels A, B, C and D. As soon as the task is assigned to a particular site, the LRM (Local resource manager) of that site accepts that task, and then assigns it to the next available computing node that meets such threshold.
In experimental results as shown in below figures 3, 4 & 5 using Task-Scheduler, we could demonstrate its viability and effectiveness in a distributed computing environment, where in the three different nodes are scheduled by scheduler.

IN OTHER WORDS, THE PROPOSED METHOD IS AS FOLLOWS

1. Tasks are randomly generated, and they differ among themselves in amount of workload,

2. Based on the group of tasks generated, Grid Scheduler record tasks' workload values MAX and MIN, according to the set of tasks given.

3. Analyzing the workload of tasks, tasks are classified into classes based on the amount of workload contained in it. Similarly, computing nodes in sites of grid platform are completely scanned, to discover values MAX and MIN, according to computing nodes' computing capabilities, so that these computing nodes are then classified.

4. Tasks in queue are presented serially to the Grid scheduler, whose function is to send task to a selected site of a grid platform, corresponding the workload and existing computing capability available in that site. If matches, this task is sent to the LRM of that site. Otherwise, the Grid Scheduler compares with next available site, according to RR (Round Robin) policy.

5. The process is repeated until all tasks inside the queue are distributed to sites, emptying completely the queue. The main objective of our experiments is to evaluate the performance of TTM over well known scheduling algorithms, First Come First Serve (FCFS) and Shortest Job First (SJF). We performed our experimentation evaluations in the heterogeneity of grid sites, through the heterogeneity and various granularities of application tasks. Experimental results of the proposed scheduling method are obtained using custom simulation model.

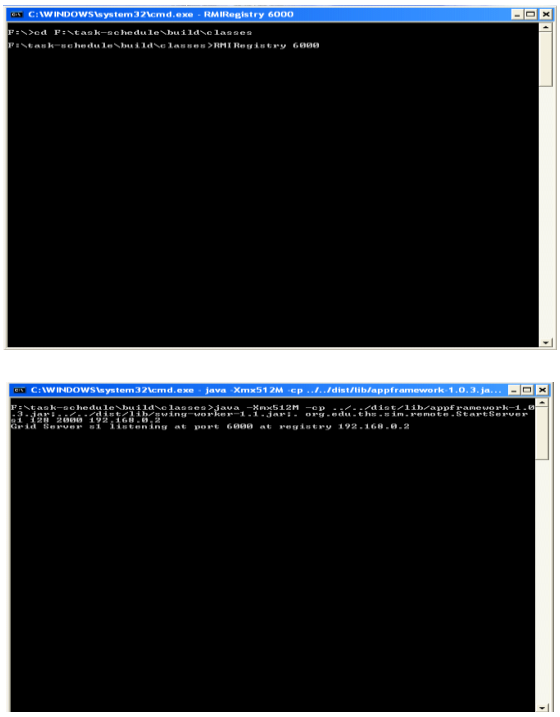


Figure-3: System1-IP Address: 192.168.0.2

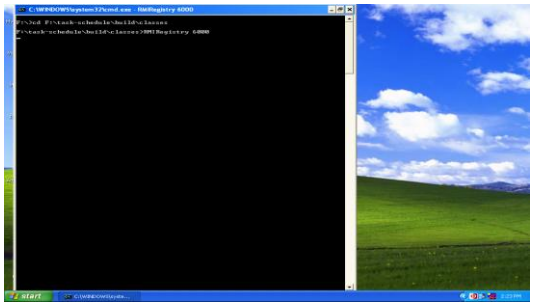
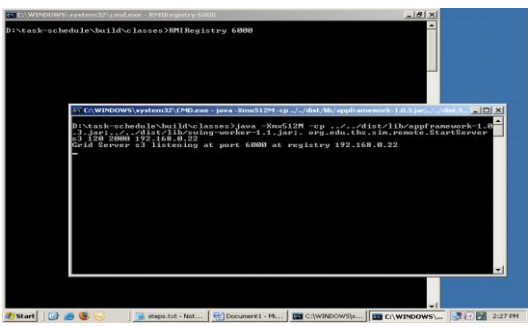


Figure 4: System2-IP Address: 192.168.0.3



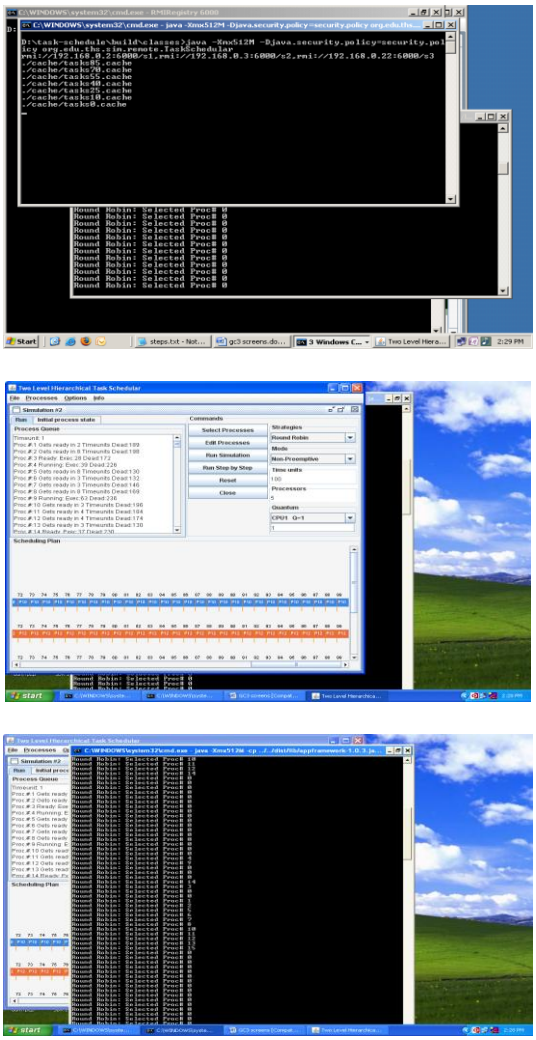


Figure 5: System3-IP Address: 192.168.0.22

V. CONCLUSION:

Advances in computing and network technologies have rapidly accelerated the development of distributed computing. Cluster computing platforms have been built by interconnecting a number of homogeneous or heterogeneous computers. Grid technology is developed aiming at the sharing of resources distributed in different geographical locations, providing large amount of computing cycles to speed up the execution of parallel applications. In this paper, we have presented a promising yet efficient scheduling method in Multiprocessor Environments, in order to provide high throughput. Through experimental results using Task-Scheduler, we could demonstrate its viability and effectiveness in such a distributed computing environment.

VI. FUTURE ENHANCEMENTS

As future work, the inclusion of task scheduling that involves communication among them can be taken up. This also means that apart from the considerations on the possibility of dependencies among tasks, the following also should be taken care so as to provide high levels of availability and reliability in the Multiprocessor Scheduling .The monitoring of computing node availability, Dynamic network traffic and Bandwidth, As well as the fault tolerance, which is very important on MP systems.

REFERENCES:

[1] H. Casanova, A. Legrand, M. Quinson, “SimGrid: A Generic Framework for Large-Scale Distributed Experiments”, in Proceedings of UKSIM 2008, The Tenth International Conference on Computer Modeling and Simulation, pp. 126 – 131, 2008.
[2] F. Dong and S.G. Akl, “Scheduling algorithms for Multiprocessor computing: state of the art and open problems”, School of Computing, Queen’s University, Technical report no. 2006-504, 2006.
[3] Y.-K. Kwok and I. Ahmad, “Static scheduling algorithms for allocating directed task graphs to multiprocessors”, ACM Computing Surveys, Volume 31, Issue 4, pp. 406 – 471, 1999.
[4] D. Menasce, D. Saha, S. Porto, V. Almeida, and S. Tripathi, “Static and dynamic processor scheduling disciplines in heterogeneous parallel architectures”, in JPDC Journal of Parallel and Distributed Computing, vol. 28, issue 1, pp. 1-18, 1995.
[5] O. Moreira, F. Valente, M. Bekooij, “Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor”, in Proceedings of EMSOFT '07 The 7th ACM & IEEE international conference on Embedded software, 2007.
[6] D. P. Silva, W. Cirne, and F.V. Brasileiro, “Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational multiprocessor models”, in Proceedings of Euro Par’2003, LNCS2790, pp.169-180, 2003.

AUTHORS

Mr. Naveen Kumar Laskari, has received the Bachelor Degree and Master Degree from JNT University, Hyderabad, India. He is Assistant Professor in the Department of Computer Science and Engineering in Hyderabad Institute of Technology and Management [HITAM], Hyderabad, A.P, India. He has presented papers and participated in number of seminars and workshops across India. His Research interest includes Grid Computing, Image Processing, Data Mining and Information Security.

Mrs. Aparna Tanam, has received her Bachelor Degree from Nagpur University, Masters Degree from JNT University, Hyderabad, India. She is an Associate Professor of Information Technology department in JB Institute of Engineering and Technology [JBIET], Hyderabad, India. Her research interest includes Grid Computing, Image Processing and Data Mining.

Mr K Ranganath, Graduated in Computer Science and Engineering from Osmania University Hyderabad, India, in 2006 and M.Tech in Software Engineering from Jawaharlal Nehru Technological University, Hyderabad, A.P., India. He is presently working as Assistant Professor in Department of Computer Science and Engineering, Hyderabad institute of Technology and Management [HITAM], Hyderabad, A.P, India. A keen research scholar and has many papers published to his credit. His research interests include Mobile Computing and Data Mining.