Load Balancing using Selection Method Grid Environment

R.Bindhuja
The Rajaas Engineering College
Vadakkangulam,
Tirunelveli Dist
Tamilnadu, India.

G.Arul Dalton
The Rajaas Engineering College
Assistant Professor/CSE
Vadakkangulam, Tirunelveli Dist
Tamilnadu, India.

Dr.T.Jebarajan
Kings Engineering College
Principal
chennai
Tamilnadu, India.

ABSTRACT
Load balancing involves assigning to each processor, work proportional to its performance, minimizing the execution time of the program. The proposed methodology is designed for task graphs that are dynamic in nature due to the presence of conditional tasks and tasks whose execution times are unpredictable but bounded. Three-phase strategy as Equal Load Balancing, Dynamic Load Balancing and Selection Method. In first phase, task nodes are mapped to the processors. In second phase, Dynamic Load Balancing is a runtime scheduling algorithm that performs list scheduling based on the actual dynamics of the schedule up to the current time. In the third phase, selection method or DLB/JR method is performed that some critical nodes are identified and duplicated for possible rescheduling at runtime, depending on the code memory constraints of the processors. The third technique provides better schedule length compared to previous two techniques, which are predominantly static in nature, with low overhead and a complexity comparable with existing techniques.

Keywords: Grid Computing, Dynamic task scheduling, Dynamic Load Balancing, Job replication.

1. INTRODUCTION
Variations in the available resources (e.g., computing power and bandwidth) may have a dramatic impact on the runtimes of parallel applications [12]. Over the years, much research has been done on this subject in grid computing. Dynamic Load Balancing (DLB) (e.g., [5], [6], [8] and [9]) and job replication (JR). DLB adapts the load on the different processors in proportion to the expected processor speeds. JR makes a given number of copies of each job, sends the copies and the original job to different processors, and waits until the first replication is finished. A comparison of the performance of those three methods on a heterogeneous globally distributed grid environment has to the best of the authors knowledge never been performed.

Each processor usually has its local memory for code storage and execution. A macro data-flow graph (Directed Acyclic Graph DAG) is used to describe an application at a high level. Mapping and scheduling of macro data-flow graphs is a fundamental and challenging problem that is being addressed by many researchers. The system model for the synthesis of a macro data-flow graph uses a high-level task graph and a processor model. Among the above models, this work focuses on the scheduling problem of task graphs having conditional nodes and tasks with unpredictable execution behavior on a set of homogeneous processors. A conditional task graph (CTG) model captures the control flow of an application, in addition to the data flow. The outgoing edges associated with a conditional node depict the control behavior of task graph at that node. During the execution of the task graph, one of the conditional branches is selected for further execution, depending on the condition evaluated at that node.

The presence of conditional tasks and tasks with unpredictable execution behavior changes the dynamics of the task graph at runtime. Thus, the initial assumptions by static schedulers about the behavior of the tasks change during execution. Hence, few processors may get relatively more free time than others and have to wait for tasks to get finished on other processors to start execution again due to the precedence constraints. This leads to load imbalances in task distribution, which may not be handled by static schedulers. Alternatively, online (dynamic) schedulers schedule tasks to processors using the information available at runtime. However, scheduling of parallel tasks often results in considerable overheads at runtime. To reduce the scheduling overhead, many low overhead techniques have been proposed for dependent tasks.

2. RELATED WORK

2.1 Equal Load Balancing

ELB parallel programs have the property that the problem can be divided into sub problems or jobs, each of which can be solved or executed in roughly the same way.

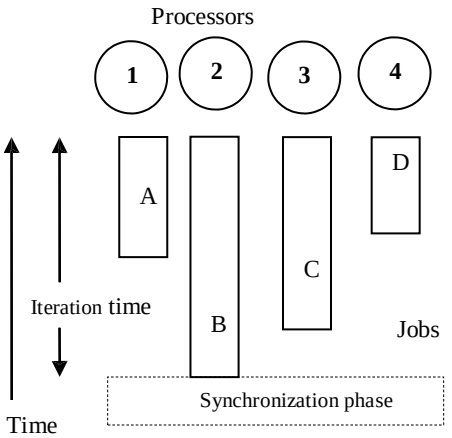

Fig.1: Equal Load Balancing

Each run consists of I iterations of P jobs, which are distributed on P processors. Each processor receives one job per iteration. Further, every run contains I synchronization moments: after computing the jobs, all the processors send their data and wait for each others data before the next iteration starts. In general, the runtime is equal to the sum of the individual iteration times (ITs). Fig. 1 presents the situation for one iteration of a BSP run in a grid environment. The fig. 1 shows that each processor receives a job, and the IT is equal to the maximum of the individual job times plus the synchronization time (ST).

2.2 Dynamic Load Balancing

DLB starts with the execution of an iteration, which does not differ from the common BSP program explained above. However, at the end of each iteration, the processors predict their processing speed for the next iteration and select one processor to be the DLB scheduler.

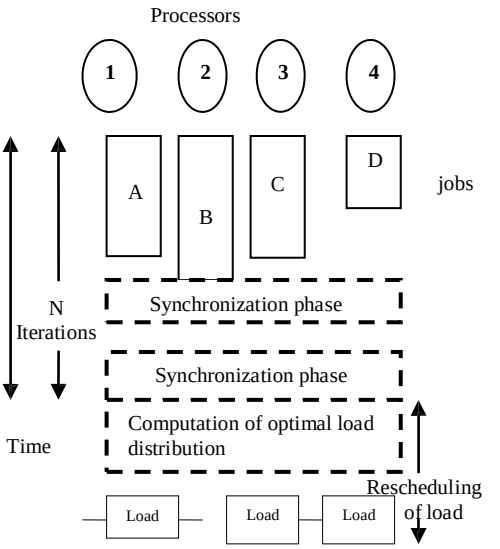

Fig. 2 Dynamic Load Balancing

After every N iterations, the processors send their prediction to this scheduler. Subsequently, this processor calculates the "optimal" load distribution given those predictions and sends relevant information to each processor. The load distribution is optimal when all processors finish their calculation exactly at the same time. Therefore, it is "optimal" when the load assigned to each processor is proportional to its predicted processor speed. Finally, all processors redistribute the load. Fig. 2 provides an overview of the different steps within a DLB implementation on four processors.

The runtime of a parallel application directly depends on the overhead of DLB, and therefore, it is better to increase the number of iterations between two load balancing steps.

2.3 Job Replication

In an R-JR (Job Replication) run, R _ 1 exact copies of each job have been created and have to be executed, such that there exist R samples of each job. Two copies of a job perform exactly the same computations: the data sets, the parameters, and the calculations are completely the same.

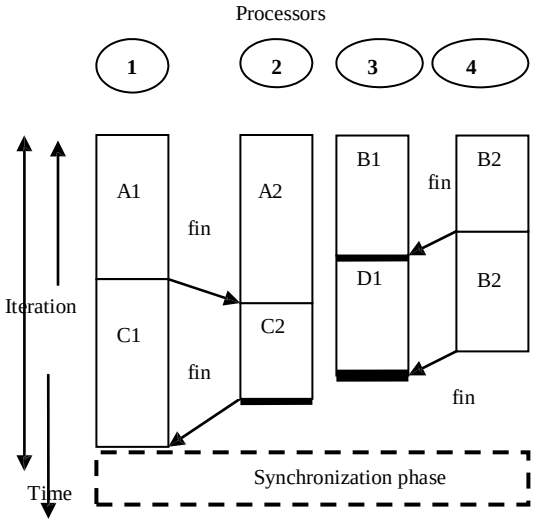

Fig. 3 Job Replication

3. PRELIMINARIES

3.1 DYNAMIC TASK SCHEDULING

In real-time, we distinguish between two types of distributed systems based on how the system is used, and its impact on the underlying infrastructure. Scheduling of real-time tasks in multiprocessor systems is to determine when and on which processor a given task executes. This can be done either statically or dynamically.

Static distributed systems are those where the processing load on the system is within known bounds such that a priori analysis can be performed. This means that the set of applications that the system could be running is know in advance, and that the workload that will be imposed on each application can be predicted within a known bound. Such systems often have a limited number of application configurations that can be executed, sometimes referred to as system modes.

Dynamic scheduling is widely employed in real-time and distributed real-time computing systems. In most real-time systems, especially distributed systems, cost-effectiveness demands that the computing system employ as much application-specific knowledge about the application and its execution environment as feasible. Much of this knowledge can be best captured in the scheduling discipline. This specification allows such application-specific scheduling disciplines to be implemented by a pluggable scheduler. An end-to-end execution model is essential to achieving end-to-end predictability of timeliness in a distributed real-time computing system. This is especially important in dynamically scheduled systems. The end-to-end execution model may be provided according to a formal standard specification (as herein), or as an ad hoc, custom-made creation by multiple different applicationprogrammers. Dynamic algorithms must be efficient; their complexity directly affects overall system performance.

3.2 GAUSSIAN ELIMINATION APPLICATION

Gaussian Elimination application is a highly communicating task graph. Since this project is focused on highly communicating task graph. A task graph is generated for Gaussian Elimination application and it is scheduled on cluster of workstations. Gaussian Elimination is a method to solve simultaneous linear equations of the form [A][X]=[C]. The goal of forward elimination is to transform the coefficient matrix into an upper triangular matrix.

In Gaussian elimination application odd level consists of single nodes. Even level consists of nodes in decreasing order starting from the order of the matrix. The total number of nodes in the m*m matrix is calculated using the formula given below,

Total number of nodes = $(m^2 + 3m) / 2$ For a 3*3 matrix, Total number of nodes=(32 + 3 * 3)/2=9

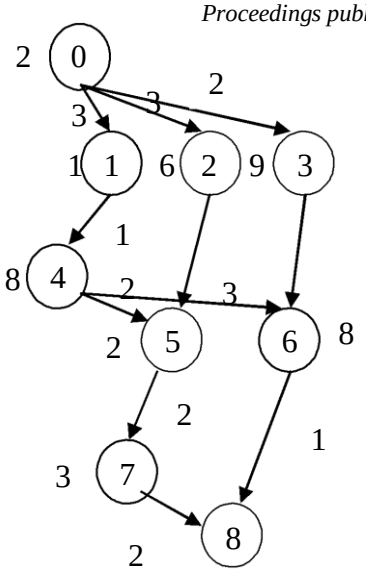

Fig 4: A 3*3 Gaussian elimination task graph with assigned computation time and communication time.

A 3*3 Gaussian elimination matrix with computation time and communication time is shown in Fig 4.

4. IMPLEMENTATION

4.1 DESIGN ARCHITECTURE

The fig. 5 shows how task is allocated, scheduled to different processor in Grid system using Selection method. Here Global scheduler accepts a task graph as its input. A task graph defines the number of tasks to be executed and also specifies their execution time, arrival time, deadline, level and criticality. The purpose of the global scheduler is to allocate tasks to different processors in the grid system.

Local scheduler is implemented to schedule the tasks for each processor and also find average utilization of each processor. When new task arrives, the scheduler makes decision dynamically by balancing the load of the processor. The new task can be rescheduled with the tasks which are in the waiting queue.

Many computational problems can be broken down into a collection of independent operations. Operations produce outputs that are used as inputs to other operations. The operations can be thought of as forming the vertices of a directed acyclic graph while the graph's directed edges show the dependencies between operations. Operations may be performed in parallel or serially, depending on when inputs become available. The Task Graph pattern addresses this problem.

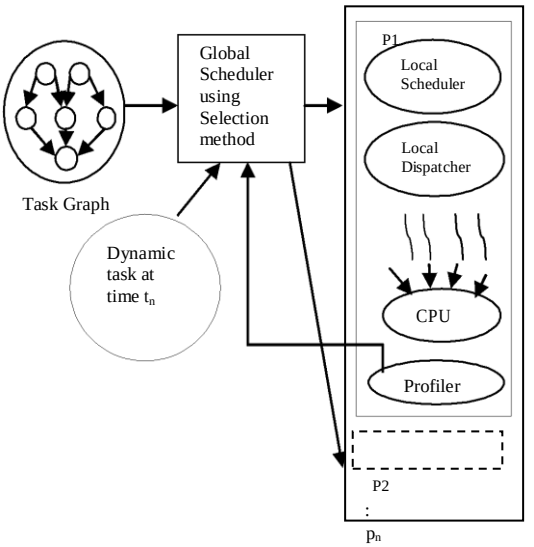

Fig 5. System Architecture

4.2 SELECTION METHOD(DLB/JR)

In this of a method that selects between the above two different types of implementations. This method has the aim to select dynamically the optimal implementation type. In this method, the processor that redistributes the load in DLB is also the processor that decides whether JR or DLB is used. When the method decides that a switch to the other type of implementation is necessary, the steps to be taken are the same as in the DLB rescheduling phase: 1) the nodes send their prediction to the scheduler, 2) the scheduler computes the optimal load distribution, and 3) the nodes redistribute their load.

An online scheduler requires each node to be mapped to every processor so that it can start a node at any available processor at runtime. Thus, the code memory requirement on each processor is the sum of all the code sizes, which is not desirable and may violate the storage memory constraints. The selection method strategy tries meeting the memory constraints on each processor by suitably

choosing nodes that are estimated to contribute most to the performance improvement during an online scheduling.

4.2.1 Dynamic Critical Path (DCP) Algorithm

The proposed DCP algorithm have the following features,

- It assigns dynamic priorities to the nodes at each step based on the dynamic critical path (defined below) so that the schedule length can be reduced monotonically.
- It changes the schedule on each processor dynamically in which the start times of the nodes are not fixed until all nodes have been scheduled.
- It selects a suitable processor for a node by looking ahead the potential start time of the node's critical child node on that processor.
- It does not exhaustively examine all processors for a node. Instead, it only considers the processors that are holding the nodes that communicate with this node.
- It schedules relatively unimportant nodes to the processors already in use in order not to waste processors.

The proposed Dynamic Critical Path(DCP) algorithm have two phases namely node selection, Processor Selection.

Pseudo code for node selection is shown below

1. Find the greatest path in the static level
2. Form a series order of the path
   Repeat :
3. Check whether all predecessor finished in the path,
   If not finished,
   If have more than one path,
   Then include in the path of order whose execution time is more.
   If exec time is same,
   Then include in the path of any order.
4. If Predecessor finished, check if any parallel task exist.
5. If so include in the path of order whose execution time is more. Until all nodes are scheduled in the DAG.

Exhaustively examining all the processors to select a suitable one can be very time consuming when the task graph is very large. Observe that the start time of a node can only be reduced by scheduling it to a processor which holds its parent nodes. And, in order to reduce the start times of the node's earlier scheduled children nodes, the processors holding such children nodes are also candidates for examination. Thus, the set of processors to be examined can be restricted to those holding the parent nodes and possibly children nodes, together with a new processor.

The brief steps of the selection method algorithm are mentioned next and will be described later:

➢ **Step 1. Construction of a concurrency graph G_C and a schedule graph G_S.** The selection method algorithm first constructs two types of graphs from the task graph and the schedule generated namely, the concurrency graph G_C and the schedule graph G_S. Fig 6 shows Concurrency graph for a 3*3 Gaussian Elimination graph shown in Fig 4.

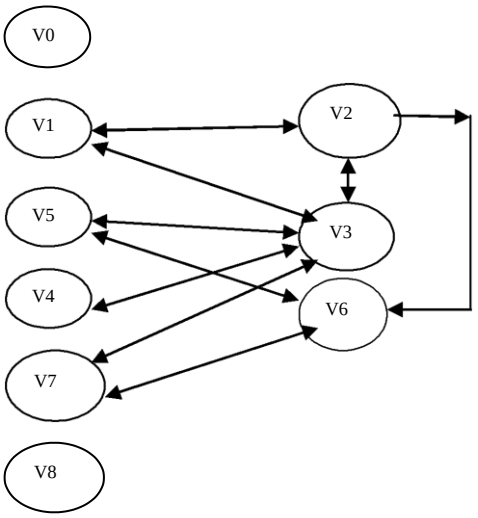

Fig 6: Concurrency Graph

These two graphs capture the possible concurrent execution in the CTG and dependency among the nodes in the schedule, respectively.

➢ **Step 2. Elimination of unimportant nodes from further consideration of replication** Using the above two graphs, identify certain nodes whose replication in other processors are not needed. These nodes are eliminated from further consideration on their replication on any of the

other processors. We define three properties for the above elimination procedure.

Property 1. If a task does not have any edge in the concurrency graph, then it need not be replicated to other processors.

Property 2. If a task vi has at most |PE|- 1 adjacent nodes in G_C and each adjacent node has a mapping in a different processor, then v_i need not be duplicated in any of the processors.

Property 3. If a node vi does not have any adjacent node v_j in G_C such that the adjacent node is mapped to the same processor, vi is eliminated from the duplication list.

- **Step 3. Weight calculation for each node processor mapping** .Each qualified node is then considered for mapping on every processor. We estimate the maximum possible start time gain that a node v_i can gain if it is mapped on a new processor by using the MinMax algorithm. A weight is assigned for each new task processor mapping vector based on the start time gain and SU of the node.

The MinMax algorithm for a schedule graph. The MinMax algorithm calculates the minimum Min_{vi} and the maximum Max_{vi} possible start time for nodes in a schedule graph.

- **Step 4. Duplication set selection.** In this final step, a subset of the node processor mapping vectors is selected for each processor so as to maximize the sum of weights within the available memory on each processor.

5. CONCLUSION

In this paper, we have made an extensive assessment that are most suitable to make parallel applications robust against the unpredictability of the grid. The selection method presented provides a powerful means to make parallel applications robust in large scale grid environments.

6. REFERENCES

[1] I. Ahmad and Y.-K. Kwok, 1994 A New Approach to Scheduling Parallel Programs Using Task Duplication.

[2] H. Attiya, 2004 Two Phase Algorithm for Load Balancing in Heterogeneous Distributed Systems.

[3] Banicescu I and V. Velusamy, 2002 Load Balancing Highly Irregular Computations with the Adaptive Factoring.

[4] S. Darbha and D.P. Agrawal, 1998 Optimal Scheduling Algorithm for Distributed Memory Machines.

[5] A.M. Dobber, G.M. Koole, and R.D. van der Mei, 2004 Dynamic Load Balancing for a Grid Application.

[6] A.M. Dobber, G.M. Koole, and R.D. van der Mei,2005 Dynamic Load Balancing Experiments in a Grid.

[7] A.M. Dobber, R.D. van der Mei, and G.M. Koole,2006 Effective Prediction of Job Processing Times in a Large Scale Grid Environment.

[8] A.M. Dobber, R.D. van der Mei, and G.M. Koole,2007 A Prediction Method for Job Running Times on Shared Processors.

[9] S. Darbha and D.P. Agrawal, 1998 Optimal Scheduling Algorithm for Distributed Memory Machines.

[10] A.M. Dobber, R.D. van der Mei, and G.M. Koole, 2006 Statistical Properties of Task Running Times in a Global Scale Grid Environment.

[11] G.-L. Park, B. Shirazi, and J. Marquis, 1998 Mapping of Parallel Tasks to Multiprocessors with Duplication.