A Hybrid Load Balancing Algorithm for Coarse-Grained Applications

AGGARWAL AKSHAI, GAN XINYU
School of Computer Science, University of Windsor, Windsor, Ontario, CANADA

Abstract: A non-preemptive hybrid load balancing algorithm is proposed for heterogeneous distributed computing environment. The hybrid model initially classifies the compute elements and jobs into different types. The algorithm maps the tasks of various jobs onto compute elements of the same type. Based on historical experience, the centralized load balancer can dynamically change the parameters in order to improve the overall performance during runtime. The algorithm balances the work load of coarse-grained applications with interdependent processes. Its working has been compared with other similar algorithms by extensive testing on a test-bed. The Hybrid Load Balancing algorithm is found to give consistently better results.

Key-Words: - load balancing, adaptive load balancing, priority based scheduling, distributed computing, heterogeneous computing environment, image processing, WebDedip, learning systems

1 Introduction

The abstract goal of load balancing discussed here is that “given a collection of tasks comprising a computation and a set of computers on which these tasks can be executed, find the mapping of tasks to computers that minimizes the run time of the computation [1]”. In a homogeneous environment, an equal partition of load will lead to load balancing. But in a heterogeneous environment, a more powerful compute-node will have to be allocated more jobs, as compared to a less powerful compute-node. Load balancing in a heterogeneous environment may mean an attempt to ensure that all the compute-nodes finish processing the jobs in the session simultaneously.

“The problem of optimal load balancing is NP complete [2]”. Heuristic approaches [3], [4] and [5] are simple but suboptimal solutions. They recalculate the algorithms’ parameters based on historical experiences. They work well under certain circumstances such as a linear combination of the n-dimensional input vector and the weight vector [5]. When considering a more general situation, it is difficult to incorporate them into a perfect mathematical model. After a certain period of time, we use collected system state information to change some of the algorithm’s parameters in order to make a better prediction of the next round of execution. For example, the processing ability of each node may vary from time to time if there is some background loads of other applications. With the updated parameters, the scheduler can make a better prediction as time goes by. For the general case, it is difficult to incorporate them into a perfect mathematical model.

Since no single scheduling method can solve all problems, we use a combination of the selected algorithms so that we may be able to take advantage of the strengths of each of the algorithms. In this paper we propose a new hybrid load balancing algorithm which can be implemented over heterogeneous distributed computing environment. The implementation requires that on each compute-element, a stand-alone java application may act as a software layer on top of the existing operating systems. The middleware takes care of both independent and inter-dependent processes. The load balancing algorithm in this paper is called a Hybrid one since it takes into account the load balancing techniques of Bhatt et.al [11] and generalizes them for heterogeneous environment. Moreover the algorithm may be used for a mini-grid system, if communication delays can be neglected.

In section 2, we present a review of different load balancing approaches and metacomputing frameworks. A detailed description of the proposed algorithms is
presented in section 3. Section 4 works out the complexity of the algorithm. The paper concludes with section 5 and references.

2 Load Balancing Algorithms

Several load balancing techniques for distributed computing environment have been presented in literature. However no single approach and framework can solve all the problems and provide really easy to use interfaces.

Load balancing strategies can be categorized into two groups: static approaches and dynamic approaches. A static approach decides the allocation of the application components before execution and can take into account both execution and communication needs [6]. A static approach works well if the application’s behavior can be accurately predicted. Dynamic load balancing is more suitable for irregular problems. A dynamic approach has to collect the system state information and make its allocation decisions at run time [6]. The chief advantage of the static policies is their simplicity. Thus there is no need to collect system state information. The disadvantage of static load balancing policies is that they cannot respond to changes in system state. Dynamic policies may provide substantial performance improvements over the static policies. However, this additional performance benefit is obtained at the cost of collecting and maintaining the system state information [7]. Adaptive algorithms are special cases of dynamic algorithms in that they adapt the activities dynamically by changing their parameters or even algorithms to match the changing system state. Consequently, adaptive algorithms can provide moderately good performance when the system state changes often and widely [5].

Some factors need to be considered when applying adaptive algorithms: correction of load profile and load prediction, trade-off of overhead and profit, and elimination of side effects in the simple execution model [9]. Simple algorithms and policies may not necessarily mean poorer performance, since complex load balancing algorithms may have an associated overhead, which may prove to be expensive. Several distributed computing environments, providing different types of tools to the developer, have been developed. Many scientists have worked on improving the efficiencies of such environments. Ease of use, is the parameter which is normally untouched by most of the scientists. WebDedip [10], developed under a research project at Gujarat University, Ahmedabad, India, is the environment which was developed with ease of use as the core theme to support HPC users like physicists, mathematicians, civil engineers, etc. Initially, the performance issues were not addressed. Later, a hybrid application centric load balancer [11], consisting of 6 algorithms, a heuristic and a normalization process, was developed for optimizing the performance.

The user’s view of a distributed computing environment should be one of a monolithic virtual machine that provides computational and data storage services. The user does not want to be concerned with the details of machines and processor type, data representation and physical location of processors and data, or the existence of other competing users [12]. The user should have the illusion of a very powerful computer on the desk. They will sit at the terminal and manipulate objects. The objects will represent data resources such as digital libraries or video streams, applications such as teleconferencing or physical simulations [13].

3 The Proposed Algorithm

The proposed algorithm is non-preemptive. The lack of the facility of migration makes the task of scheduling simpler and may not lead to a worse performance in most of the cases. The hybrid algorithms is based on [11]. However the algorithm of [11] does not refer to a heterogeneous environment. The proposed algorithm provides the facility of load balancing to a heterogeneous set of compute-nodes, connected in parallel. The algorithm divides the compute-nodes into a global task scheduler or server and worker-nodes. The server receives all the applications into a global task scheduler or server and worker-nodes. The server receives all the applications, works out the mapping of the processes to the worker nodes and inter-acts with the user.

3.1 Preliminary Definitions

TNLB will be the maximum value of the execution time of all the workers nodes [14]:

$$\text{TNLB} = \max T_i , \quad 0 \leq i < n$$
where $T_i$ is the elapsed time of worker $i$. If $W_i$ is the amount of workload assigned to worker $i$ and $P_i(t)$ is the processing capability of worker $i$ at time $t$, then

$$\int_0^{T_i} P_i(t) dt = W_i$$

$W_i$ is the total work load of worker $i$ which includes background load. In practice, $P_i(t)$ is normally not known in advance.

### 3.2 The Assumptions for the Proposed Model

The meta-data about the applications including the process run time for each process and the interdependency relationships, through the interdependency graph, have to be known. We can also classify the workers nodes into several classes based on the capability of the machines [15].

The global task scheduler decides task allocation and also collects processing information from the worker nodes. It also informs the user about the status of his job.

The state information may be stored as a table with the following columns in the server:

- Node ID
- Finished Processes (with the actual computing time)
- Running Processes (with the estimated computing time)
- Wait Queue

We assume that the jobs follow the Poisson arrival rate.

Based on GR.batch [16] Model, we changed the node processing throughput to service rate using standard process execution time. The GR.batch Model does not specifically consider the process interdependency relationships while the proposed model will take that into account. The standard process we choose is Linpack [17] benchmark. We normalize our node types and classes of sub task as follows:

Suppose, there are $m$ different node types. Each node type $T_i$, where $i \in M = \{1, 2, ..., m\}$, is represented by a 3-tuple $(T_i, \mu_i, \psi_i)$ where $\mu_i$ and $\psi_i$ are the service rate and the number of nodes of type $T_i$, respectively.

The processing node composition in the system can be represented as a set of 3-tuples:

$$\{(T_{i_1}, \mu_{i_1}, \psi_{i_1}), (T_{i_2}, \mu_{i_2}, \psi_{i_2}), \ldots\}$$

The total number of nodes in the system

$$\Psi = \sum_{i \in M} \psi_i$$

The node IDs of type $T_1$ are $(N_{i_1}, N_{i_2}, \ldots, N_{i_{\psi_1}})$, and those of type $T_2$ are $(N_{i_1 + \psi_1}, N_{i_1 + \psi_1 + 1}, \ldots, N_{i_1 + \psi_1 + \psi_2})$, and so on. $T(N_i)$ is used to refer the node type of a processing node having ID $N_i$. Relative service rate of node type $T_i$ with respect to node type $T_j$ is denoted as $\mu_j^i$.

$$\mu_j^i = \frac{\mu_i}{\mu_j}$$

We define $\mu_i = 1/TS_i$ where $TS_i$ is the time to execute the standard process (Linpack) on node type $i$ and $\mu_i$ is the service rate of node type $i$. We denote $\lambda$ as the application arrival rate. Each application follows Poisson distribution and the standard Poisson arrival rate will be multiplied by $\lambda$. Application arrival rate is defined by the user and stored in a configuration file (appinfo.txt). Application tasks are categorized into $n$ classes. Each task class $C_i$ where $i \in J = \{1, 2, ..., n\}$, is represented by two attributes:

- Service Demand, denoted as $\omega_i$, is the processing requirement of a class $C_i$ task.
- Code Length, denoted as $l_i$, is the length in Kbytes of the task transfer message generated for a class $C_i$ task if assigned remotely. The task class composition can then be represented as a set of 3-tuples:

$$\{(C_{i_1}, \omega_{i_1}, l_{i_1}), (C_{i_2}, \omega_{i_2}, l_{i_2}), \ldots, (C_{i_n}, \omega_{i_n}, l_{i_n})\}$$

We may run a user’s process of a job on a standard machine and get the execution time. This execution time is defined as the service demand for the process. The service demand of a process is an estimated one and may not be very accurate. However, it can be tuned dynamically.

If $T_S$ be the time to execute the standard process on a standard machine and if $T_U$ be the time to execute the user process on a worker node,
\[ \omega = \frac{\text{TU}}{\text{TS}} \]

where \( \omega \) is the service demand of the user process. We also use the standard process (Linpack) as the basic unit of a process. The service rate is defined only for the processes within the target application. In case there is any background load, the service rate for our interested processes may be deducted accordingly.

In this way, we can compare the service demand and service rate over different machines so that:

Time needed to complete a task is \( \frac{\omega_i}{\mu_i} \).

We assume that all the user’s jobs will arrive at the server before the central scheduler can distribute those jobs to the worker nodes. So far we can define the actual work load and virtual work load as follows:

Actual work load:

\[ W_j = \sum_{i=1}^{k} \frac{\omega_i}{\mu_j} \]

Virtual work load:

\[ VW_j = \sum_{i=1}^{q} \frac{\omega_i}{\mu_j} + \sum_{i=1}^{k} \frac{\omega_i}{\mu_j} \]

where \( k \) is the number of processes in the Ready-to-run Queue and \( q \) is the number of processes in the Wait Queue.

Average work load of one node:

\[ AW = \frac{\sum_{i=1}^{9} VW_i}{\psi} \]

The system will recalculate \( \mu_i \) after it finishes processing \( h \psi \) processes, where

\[ \mu_i^{k} = \frac{ET_i}{AT_i} \mu_i^{k-1} \]

\( AT_i \) is the average actual processing time of node \( i \) and \( ET_i \) is the average estimated processing time of node \( i \). \( \mu_i^{k-1} \) is the previous service rate value of node \( i \).

\( h \) is the algorithm parameter (an integer) that determines the interval time to collect system state information.

3.3 System Overview

From figure 1, we can see that communication daemon and ftp server should be installed on every node of the system. The server has the whole picture of the application status. The central scheduler on the server side is responsible for allocating processes to workers according to current work load information.

Estimated processes’ computing time is used as the metric of work load.

![System Overview](image_url)
3.4 The Main Algorithm

Server Side:
1. Generate the workload and make them arrive in Poisson arrival rate.
2. Put the applications in server queue (squeue.txt).
3. Find out the processes’ interdependent information of each application and put them into server process queue (swq.txt).
4. Calculate each workers’ virtual work load.
5. Newly arrived processes (during run time) are allocated by the central scheduler to the most light-loaded (Virtual Work Load \( \psi_j \)) node’s Wait Queue according to FIFO [38] policy.
6. Processes with no parent processes will be put directly in the Ready-to-run Queue. Other processes will be put in the waiting queue of the worker.

Worker Side:
1. When a process in Ready-to-run Queue finishes its job, the intermediate result is saved locally.
2. When all the processes in Ready-to-run Queue finish their job, the worker will put the first process of Wait Queue to Ready-to-run Queue.
3. Whenever a process at the front position of the waiting queue cannot be running because of the interdependency restriction, it will notify the server and the server looks for the parent processes.
4. Worker recalculates its process capability periodically and informs the server.

4. Complexity of the Algorithm

The complexity of the hybrid algorithm is \( O(n) \). \( n \) is the number of applications in the server queue. There is a single loop in the program which depends upon the input size of the applications.

There are other factors which affect the complexity:
1. Disk accessing time will increase when the amount of data in the intermediate results increase.
2. The communication overhead will increase when the number of nodes in the system scales up.
3. It is also more difficult for the server to maintain the system state information and coordinate workers when the number of workers increases.
4. Unpredictable Internet traffic may conflict with the messages we use to transfer system state information. Due to the nature of Ethernet connection, the program will try to send the identical message again if it fails to transfer. The number of retries is also unpredictable. The above factors depend upon the traffic of the network, the number of worker nodes and the nature of applications. Other costs within the program remain constant.

5. Conclusion and Future Work

A new hybrid load balancing algorithm has been proposed. It is based on the Application-centric load balancing algorithm of Bhatt et. Al [11]. However this algorithm is able to map interdependent tasks to a heterogeneous system of compute-nodes. Estimated processing time is used as the metric of the workload. It automatically takes care of the process interdependency relationships and background load. Algorithm parameters are tuned during runtime in order to make a better prediction for the next round of execution. It is found that the hybrid algorithm performs relatively better for both independent and interdependent applications.

Easy to use metacomputing systems which can transparently manage all the computing resources over the Internet is the goal of all grid frameworks. Such frameworks use some bandwidth measurement tools to improve the overall performance when communication delays are present. We are working on generalizing the Hybrid load balancing algorithm to make it usable on a grid for improving the utilization of compute and communication resources and for reducing the session completion times.

References:


