FBLB: A Feedback Based Scheme For Scheduling Medical Post Processing Applications In Clusters

Ravi Madipadaga, Sabi Shaw
Distributed Systems Group
Siemens Corporate Technology
Bangalore, India.

Saikat Mukherjee
International Institute of Information Technology
Bangalore, India.

Abstract: A new architecture for scheduling an open-ended set of medical post processing applications in clusters is described in this work. The scheduler takes into consideration the characteristics of the set of currently executing applications as well as the incoming request to ensure optimal application performance. The approach uses a feedback mechanism to learn the resource requirements of an application and is non-intrusive. The scheduler is not tightly coupled with the applications and therefore can schedule an open-ended set of applications.

Key–Words: Scheduling, Clusters, Medical Post Processing, Load Balancing, Application Profiling

1 Introduction

Scheduling client requests by load balancers in traditional enterprise computing clusters is significantly different from scheduling requests in a medical diagnostic and post-processing cluster. The primary difference is in the characteristic of the tasks performed to cater to the user request. In enterprise computing, tasks are usually short-lived and require almost the same amount of computational resources and hence simple load balancing algorithms like round-robin are well suited to such an environment. Medical diagnostic and post-processing applications are long-lived, cannot be migrated to another system at runtime (due to the prohibitive volume of data involved) and vary significantly with respect to the amount of computational resources required. Besides, such applications are open-ended and are introduced into the cluster over time. Hence there is a need for the scheduler to learn the characteristics of the applications. Therefore simple scheduling algorithms are not suited to such an environment. This paper suggests a new online scheduling architecture FBLB: Feed-Back Load Balancing based on application profiling. Applications are profiled at run time and their resource usage profiles are logged. Simple statistical techniques are then employed over the logs to predict the application’s resource affinity and usage. In FBLB, the scheduler maintains a list of applications currently executing on the cluster. It then calculates the load on the each node, not only at a given instance but at a given time on the future as well. This is significantly better than using system performance counters which are very dynamic in nature and provide immediate load information measures only. The scheduler then uses the resource usage pattern along with the history of scheduled applications to schedule the application on a suitable cluster node.

Preliminary tests on a cluster with a representative set of medical applications are very promising and show very significant performance improvements over other techniques.

2 Related Work

Scheduling based on instantaneous load measurements in cluster nodes [1, 2] do not provide optimal scheduling. Application developers using Apple’s agents [3] need to conform to the Apple’s agent rules, which tightly couples the application to the scheduler. Also in Apple’s, each application needs a customized scheduling agent thereby making it unsuitable for open-ended applications. In FBLB, application independent agents determine the resource usage of an application. This information is fed back into the scheduler, thereby enabling it to schedule new applications. Process migration techniques require processes to be rescheduled to a different node to balance the load in clusters [4, 3, 8], however since medical applications usually work on huge data sets, process migration is not a feasible option for schedul-

---

1 as observed in Section 7.
ing such applications. Prophet [5] is used for applications where data can be decomposed and processed on different nodes in the cluster. PBLB [7] tries to schedule fractions of a problem into the different nodes taking into consideration the non-linearity with respect to problem size. Scheduling using Mehra’s algorithm [6] requires the application to be run on nodes, under various loading conditions (using synthetic workload generators). This requires the application to be run offline, thereby making it unsuitable for applications which are introduced directly into the cluster. Also the accuracy of scheduling degrades for long running jobs.

3 System Architecture

The system architecture of FBLB is shown in Figure 1. The main components of the system are explained below.

3.1 Monitors

The monitors are agents on each node in the cluster. The monitor on a node observes each application instance’s interaction with the system unlike others [5, 7, 4, 6] where the overall system load of specific resources is monitored. The data collected from application performance counters at specified time intervals is pushed into the log manager. The performance counters monitored by the monitor are described in Section 4.

3.2 Log Manager

The log manager accepts data from all the monitoring nodes and stores it in a database. Data from different applications are maintained in separate tables. The log manager can also handle concurrent instances of the same application. This arrangement enables storage of large volumes of profile data, which is used by the distiller to accurately determine the characteristics of the application.

3.3 Distiller

The distiller uses the logs in the log manager to calculate an application’s footprint or what we refer to as its profile. The distiller generates a dictionary which is a hash function with the application name being the key and the application’s profile being the value returned. An example dictionary in tabular format is shown in the table below. The distiller’s operations are explained in Section 5.

<table>
<thead>
<tr>
<th>Application</th>
<th>Mem (Mb)</th>
<th>CPU</th>
<th>Execution Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CT Viewer</td>
<td>2800</td>
<td>0.7</td>
<td>6000</td>
</tr>
<tr>
<td>Hanging Protocol</td>
<td>500</td>
<td>0.6</td>
<td>50</td>
</tr>
<tr>
<td>Composer</td>
<td>450</td>
<td>0.8</td>
<td>120</td>
</tr>
</tbody>
</table>

Sample Dictionary

3.4 Scheduler

The scheduler distributes the incoming client requests among the nodes in the cluster. The scheduler uses the dictionary provided by the distiller to determine the requested application’s resource requirements and then uses an online scheduling technique (Section 6) to schedule the request on an optimal node in the cluster.

4 Counters Logged by the Monitors

The following counters were found useful in determining the footprint of the application.

4.0.1 Memory

In any paged system, the physical memory footprint (the working set) of the application is the most significant as far as application performance is concerned. This was confirmed via experimentation.

\(^2\)The log manager is not required once the profiles of all the applications are available.
5 Determining the Footprint of an Application

To determine the footprint of an application, the distiller has to perform the following tasks:

- Calculate the application’s resource utilization for a particular run
- Incorporate this run’s resource usage into the application’s resource profile history (update the dictionary)

5.1 Calculating an application’s resource utilization

Once an application has completed its execution, the distiller computes a single value or a metric for each resource. This metric reflects the application’s usage of that particular resource for the run. The counters being monitored are orthogonal in nature and have differing impacts on the application’s performance. Therefore a single value combining all these counters is not recommended, unlike as attempted in [2].

5.1.1 Estimating the CPU metric

The CPU metric is calculated with the following formula:

\[ M_{cpu} = \frac{1}{T} \sum_{k=0}^{N} \frac{\phi(k) \cdot \Delta(k)}{100} \]  

where, \( M_{cpu} \) is the CPU metric, \( \phi(k) \) is the percentage processor time used by the application, \( N \) is the total number of samples, \( \Delta(k) \) represents the sampling duration for the \( k^{th} \) sample (in case of non linear sampling) and \( T \) is the execution time of the application.

When the systems are heavily loaded, the monitors reduce the rate at which it samples the performance counters. The above equation takes into consideration unequal sampling durations and calculates the average processor usage of an application instance.

5.1.2 Estimating the memory metric

The memory metric is calculated with the following formula:

\[ M_{mem} = \text{Median } \theta \]  

where, \( M_{mem} \) is the memory metric and \( \theta \) is the set of working-set values sampled over the run of an application instance. The median value is used to capture the amount of memory the application uses for most of its execution time. Using the average value here will lead to unnecessary dilution of the amount of memory used by the application.

5.2 Dictionary Updates

Once the distiller has calculated the metrics for a particular run of an application, the application’s profile in the dictionary is updated. To update both the CPU and memory metrics for an application in the dictionary, the following formula is used:

\[ M_{cntr}(n + 1) = \alpha \times M_{cntr}(n) + (1 - \alpha) \times m_{cntr} \]  

where, \( M_{cntr}(n + 1) \) is the new consolidated metric for a counter \( cntr \), \( M_{cntr}(n) \) is the old consolidated metric, \( m_{cntr} \) is the newly estimated counter metric from the application’s latest run. \( \alpha \) is a constant which determines the effect of the newly concluded metric value on the consolidated metric value in the dictionary. \( M_{cntr} \) is set to some default value initially.

6 Scheduling strategy

To serve a client request, the scheduler looks up the dictionary to determine the profile of the application being requested. It then calculates the memory load and CPU load on each node. To estimate the memory load and CPU load on a node in the cluster, the scheduler maintains a list of applications currently executing on each node of the cluster (called the Applist). The memory load and CPU load values are dependent on the profiles of the applications executing on the particular node and are calculated using the following equations:

\[ MemoryLoad_j = \sum_{i=1}^{N} M_{mem}(App_i) \times \delta(App_i) \]  

where \( N \) is the number of applications running on node \( j \). \( M_{mem}(App_i) \) is the memory metric of application \( i \) and \( \delta \) is the intersection time. The intersection time is defined as the time overlap between
the newly requested application’s execution time and the currently running application’s remaining execution time.

For instance, if a currently executing application’s remaining execution time is estimated to be \( t_1 \) seconds and the newly requested application’s estimated time (from the dictionary) is \( t_2 \) seconds, \( \delta \) is calculated as,

\[
\delta = \begin{cases} 
  t_2; & \text{if } t_1 \geq t_2 \\
  t_1; & \text{if } t_1 < t_2
\end{cases}
\]

Similarly, for calculating the CPU load, the following formula is used,

\[
CPULoad_j = \sum_{i=1}^{N} M_{CPU}(App_i) \times \delta(App_i)
\]

where \( N \) is the number of applications running on node \( j \). \( M_{CPU}(App_i) \) is the CPU metric of application \( i \) and \( \delta \) is the intersection time.

The memory load along with the CPU load gives the scheduler an estimate of the load on a particular node in the cluster. Since all the applications are scheduled through the scheduler, it can accurately estimate the CPU and memory loads.\(^3\)

In our experiments, it was observed that, executing an application on a node where the available memory is much greater than the required memory does not speed up the application’s performance. However if the available memory in the node is less than the memory required by the application, the performance of the application severely degrades due to paging [2]. Hence before scheduling an application on a node, the scheduler ensures that there is enough memory available on the node. It was also observed that the performance of the applications were directly dependent on the load on the CPU.

The scheduler selects the node with the least loaded CPU from the set of nodes which have sufficient memory available for the application. If no such node exists, the scheduler then schedules the application on the node having the maximum available memory. The pseudo code for the scheduling algorithm is given below.

```plaintext
//initialization:
CPULoad_{min} = \infty;
SelectedNode = -1;
N = total number of nodes;
sort(CPULoad)_{nodes};
sort(TotalMem - MemLoad)_{nodes};
for j=1 to N;
  if(CPULoad_{min} > CPULoad_{sorted_j} and
      Required Mem < (TotalMem_j - MemLoad_j))
    CPULoad_{min} = CPULoad_{sorted_j};
    SelectedNode = j;
if(Selected Node == -1)
  Selected Node = (TotalMem - MemLoad)_{max} node;
```

Scheduling Algorithm

Once the scheduler dispatches an application for execution on a particular node, it appends an entry to the Applist of the node. It should be noted that when an application finishes its execution, it is automatically removed from the Applist.

7 Results

The scheduling algorithm was compared with other approaches and showed significant improvement in scheduling quality.

7.1 Test Environment

The test environment consisted of three server machines, which were heterogeneous in capabilities. The memory sizes on the server machines ranged from 7GB to 16GB. The servers were connected through a Gigabit switch. Four clients were included in the test setup. One client was designated as the log manager and distiller node. Though not necessary, this was done to minimize the interference with measurements.

7.2 Test Procedure

The tests were performed with a set of three applications:

- **CTViewer**: This application is a standard 4D viewer (movie of 3D volume rendered data). The data set used was in the order of 3-4Gb.
- **Hanging Protocol**: This is a simple 2D viewer application with small data sets.

\(^3\)Applications being executed on the cluster for the first time are executed with default profile values. At the end of the first run, the distiller will have the actual profile of the application, which is used for further calculations.
• Composer: This application is a CPU intensive application performing a large number of numerical calculations.

The test runs were automated by batch scripts that activated these applications in a predetermined sequence. The Hanging Protocol and Composer applications were smaller applications, which were fired up frequently with numerous instances running concurrently. Each client then requests a CTViewer application to be launched. The test scripts were run using three different scheduling techniques;

• Online or Dynamic Scheduling: The load on the cluster nodes is determined using the system performance counters at the time of scheduling the application. The application is scheduled on the least loaded node.

• Resource Corridor: Due to the requirement of smooth execution predictability, the approach of resource corridor (RC) was initially attempted. In RC, each compute node is divided into logical nodes of asymmetric capacity, with predetermined sizes. This division is based on a rough classification of the targeted applications. To each logical node only a single appropriate application is scheduled, thus bringing down the problem of online scheduling to a best fit search. But since the division into logical nodes is done empirically, the scheduling quality is suffered.

• FBLB: The feedback based approach described in this paper.

7.3 Experimental Results

• Hanging Protocol: The graph (Figure 2) shows the completion times for the Hanging Protocol application. The average execution time of applications scheduled using FBLB is significantly less than the other two approaches.

• Composer: The graph (Figure 3) shows the completion times for the Composer application. It shows the feedback mechanism achieves better results.

• CT Viewer: The graph (Figure 4) shows the frame rates achieved for the CTViewer application. Higher frame rates indicate better results. Here we see that majority of the Feedback mechanism (FB) values are higher than the Resource Corridor (RC and Dynamic).

8 Conclusion and Future Directions

The results of the feedback based application profiling and scheduling of medical applications shows promising results over other techniques. The main advantage of the approach used in FBLB is that after some runs, the distiller latches on accurately to the actual profile of the application. Also, because the profile is determined by taking into consideration all the previous runs of the application, occasional deviations in the resource usage patterns do not affect the resultant profile significantly. As the scheduling is done based on the observed application profile, over time statistically
optimal scheduling is achieved. However there are still some areas where further research can improve the scheduling quality;

- **Disk and Network I/O:** Disk and network adapter usage in applications also need to be monitored and the scheduler should take into consideration these resource usage parameters into its scheduling scheme.

- **Interactivity:** Some medical applications are interactive in nature and hence the techniques described in this work will not be sufficient. Preliminary work is in progress on algorithms to take into consideration the interactivity of applications.

- **Variability in Data Input:** Depending on the application characteristics, it is not possible to predict apriori, the data set the application is going to load into the memory. However, in some specific applications, information about the data set to be used is available upfront as workflow information (from the client’s request). The load balancer can use this information to predict accurately the memory requirements for the application.

- **Application priority Modeling:** Not all applications in the medical domain have equal priority. Some batch applications, can be executed; when the load on the cluster is less, whereas post processing applications need to be executed instantaneously. By delaying/temporarily stopping the low priority jobs, resources can be freed up for high priority jobs.

**References:**


