Load Balancing in a Distributed Processing System for High-Energy Physics (UFMulti)

Jagadeesh Kasaraneni
Theodore Johnson
Dept. of Computer and Information Science
University of Florida
Gainesville, Fl 32611-2024

Paul Avery
Dept. of Physics
University of Florida
Gainesville, Fl 32611-2024

Abstract

Experiments in High Energy Physics (HEP) generate tremendous amounts of data. For example, the accelerator at CERN is expected to generate petabytes per year. New HEP discoveries require that the experimental data be carefully analyzed to identify the events of interest. To take advantage of distributed processing, we have developed a package, UFMulti, which allows physicists to specify and execute a complex distributed application. In this paper, we report on the load balancing module of UFMulti. Our algorithm takes advantage of the particular structures of typical HEP computations to perform very effective load balancing. For example, our dynamic load balancer takes advantage of buffered data to partially allocate a processor between tasks. As a result, it often provides better performance than any static allocation of processors to tasks.

1 Introduction

We developed the UFMulti system to automate the distributed processing of High Energy Physics (HEP) data. In this paper, we discuss the load balancing that we implemented in UFMulti. A brief overview of the High-Energy Physics problem and how UFMulti helps solve this problem are discussed below followed by a section explaining the need for the load balancing in UFMulti.

1.1 HEP applications

A High-Energy Physics (HEP) application needs to process large volumes of data composed of millions of *events* (i.e., experimental observations from a particle accelerator). The events consists of kilobytes to megabytes of data, and represent the data collected from colliding two subatomic particles. This data is generated either experimentally or by Monte Carlo simulations. Processing each event takes anywhere from several milliseconds to several hours; consequently, the running time for the entire data can range from several hours to several days without parallel processing.

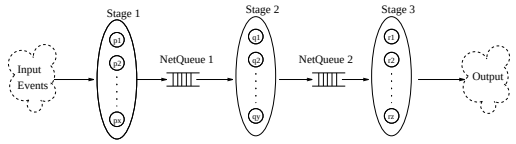

Figure 1: An HEP application divided into stages glued together by NetQueues

We take advantage of two important characteristics of typical HEP jobs. First, events are independent, so all events can be processed in parallel. Second, the analysis performed on events can be broken into coarse grained stages. The analysis at each stage is typically written by different teams of scientists. In PASS2 processing, the first stage reconstructs an events (i.e., calculates particle tracks, etc.) from the raw data, and the second stage makes a preliminary examination of the data to determine if the event in "interesting". Since the programs are written by different teams, it is convenient to treat each stage separately, and only integrate the processing at run time. For more details on HEP processing see [2].

1.2 UFMulti

UFMulti is a toolkit designed for distributing the components of an HEP application across multiple Unix workstations [1]. UFMulti considers every application to be composed of a number of interacting stages, each stage consisting of one or more tasks which are copies of an user analysis program. Each task can potentially be executed on a different workstation, thus providing for both distributed and parallel computa-

tion of a stage.

Stages interact with each other by passing processed events from one stage to another stage. This "event passing" is achieved by using a queue abstraction called NetQueues. A NetQueue process is an intelligent buffer with limited space. Multiple tasks can be dequeueing from and enqueueing into the NetQueue simultaneously, although these I/O requests are processed serially. The NetQueue blocks the enqueuers (or dequeuers) when the buffer is full (or empty). The enqueuers/dequeuers are unaware of this blocking.

For example, Figure 1 shows an HEP application divided into three stages: Stage1, Stage2 and Stage3. Each stage has a set of tasks running on different workstations - p1, p2, ..., px in Stage1, q1, q2, ..., qy in Stage2, and r1, r2, ..., rz in Stage3. The tasks of a stage are instances of the same program. After performing analysis on an event, tasks of a stage enqueue the processed event into a NetQueue, from which the tasks of the next stage dequeue. These analysis tasks need not be aware of where the NetQueue resides or its implementation details. They only need to know how to enqueue into and dequeue from the NetQueue.

1.3 Need for Load Balancing

The real world system constraints (like the number of workstations available, the network bandwidth, and etc.) impose an upper limit on the total number of tasks that can run in all the stages of the system. This number is to be divided among the stages depending on the event processing capabilities of the tasks. The rate at which a stage processes the events depends on the number of analysis tasks given to that stage and is the sum of the event-processing rates of those tasks. Thus, one can either increase or decrease the event processing rate of a stage by increasing or decreasing the number of analysis tasks running at that stage.

The throughput of an HEP application (such as that shown in Figure 1) is limited by the stage which processes the events the slowest. This stage is the bottleneck of the system. Because of the bottleneck stage and the limited buffer space in the NetQueues, the other stages of the system will not be able to process the events at the rate they are capable of.

Without automatic load balancing, the user needs to guess the event processing capabilities of the tasks at each stage and allocate the number of tasks accordingly. The event processing rate of the bottleneck stage and the system (and, thereby, the job completion time) depends on this guess. In addition, the user needs to monitor and redesign the system, while the application is running, so that it can adapt to the changing event processing rates of the analysis tasks.

In this paper, we describe the load balancer that we built into UFMulti. We show that it produces an optimal static system design. We also implemented a dynamic load balancer. The dynamic load balancer

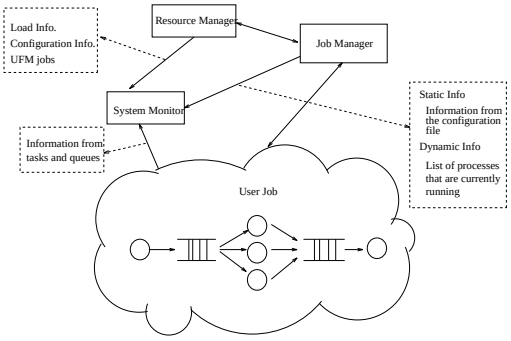

Figure 2: Interaction of the System Monitor with other components

takes advantage of the buffering in the system to allow a partial allocation of a processor among different stages. As a result, dynamic load balancing usually gives higher performance than any static allocation of processors to stages.

The load balancing component of UFMulti makes use of a system monitor to detect imbalances, and a load balancer. We describe both components in the next two sections.

The development of the load balancing module is done in two parts. In the first part, we developed System Monitor. System Monitor collects the required information from the job and displays it to the user. Using this information, the user can balance the stages of the job manually by moving processors from one stage to another stage. These steps are automated in the Load Balancer which is developed in the second part.

Much research has been done in designing load balancing algorithms for distributed systems. Readers interested in the taxonomy of the load balancing algorithms should refer to Casavant and Kuhl[5] and Wang and Morris[4].

2 System Monitor

System Monitor provides the user with a monitoring tool and a manual load balancing mechanism. The objective of the monitoring is to find the status of the job and identify bottlenecks, if any. The monitoring part of this tool is also used in the load balancing module which is developed later.

The interaction of the System Monitor with the other components of UFMulti tool kit is shown in Figure 2. The components are in solid boxes. The solid arrows represent the direction of flow of communication. The dashed boxes represent the information that is being obtained. The two principle components of UFMulti are the Job Manager and the Resource Manager. The Job Manager is the control process for a single job. It starts and configures a job, triggers statistics collection, and celanly terminates the job. The Resource Manager keeps track of resource use by the

different jobs that are executing, and provides the Job Manager with lists of lightly loaded resources.

The System Monitor communicates with the Resource Manager and the Job Manager to obtain the information displayed to the user and used by the load balancer. The information that is collected includes task processing rates, item processing rates (number of events processed per second by a task), queue lengths (in the Netqueues) and the percentage of time that tasks are idle.

The user can manually balance the system by watching the system monitor and using a facility to re-assign processors to tasks. Intuitively, the user should monitor the system and watch for full or empty queues, which indicate an imbalance. Then, based on task processing rates, the user can make a re-allocation. These steps (adjusting the monitoring interval, detecting the point of imbalance, and moving tasks from one stage to another stage to balance the system) are automated in the Load Balancer.

3 Load Balancer

We make two assumptions about the job layout. First, the job layout is assumed to have a linear structure as shown in Figure 1. All the stages except the first and the last are connected to two queues. Second, the general functioning of all the tasks is to repeatedly dequeue an event, perform analysis on the event, enqueue it into the next queue. The only exceptions are the first stage and the last stage. The first stage does not dequeue and the last stage does not enqueue.

We make some definitions for use in our discussion. A *system* is a collection of sonnected stages of a job. The *start* is the first stage and the *sink* is the last stage. The *event processing rate* of a task is the rate at which it can process events (i.e., number per second). The event processing rate of a stage is the sum of the event processing rates of the tasks in the stage. The *bottleneck stage* is the stage with the lowest event processing rate.

A high level description of the load balancing algorithm is given below:

Load Balancing Algorithm:
    1) perform Static Re-allocation
    2) perform Dynamic Load Balancing
Static Re-allocation:
    1. Collect event processing rates.
    2. Design a system which has the lowest completion time.
Dynamic Load Balancing:
    1. Collect event processing.
    2. If the job has finished, exit.
    3. Search for a point of imbalance.
    4. Balance the system around that point.
    5. Go To step 1

Static Re-allocation sets up a reasonably intelligent system which is monitored by the next phase, dynamic load balancing. Synamic load balancing ensures that the system remain in balance.

3.1 Phase 1: Static Re-Allocation

After the Job Manager starts up the system, it starts the Load Balancer process. The Load Balancer collects the event processing rates of all the tasks and enters the static re-allocation phase. The objective of this phase is to redesign the system in such a way that yields the best completion time for that job. While redesigning the system, the tasks are allocated to the stages inversely proportion to their respective event processing rates. This new design is imposed on the system. Note that the new design may be the same as the old one in which case the system is not changed.

Static reallocation produces an optimal static design, but the performance may be suboptimal, for two reasons. First, the event processing rates can change if the characteristics of the events changes. Second, the load on the workstations may change (due to non-IFMulti jobs), changing their event processing rates. Even if event processing rates do not change, the static reallocation is likely to be suboptimal, because fractions of a processor can't be allocated to a stage. Dynamic load balancing is needed to react to changes in system characteristics, and to perform fractional processor allocation.

3.2 Phase 2: Dynamic Load Balancing

After Re-allocation the Load Balancer enters the dynamic load balancing phase. In this phase, the Load Balancer repeatedly monitors the system, detects a point of imbalance (if any), and takes steps to balance the system around that point. The point of imbalance is a queue which is either full or empty. The point of imbalance divides the system into two parts, system on the left and system on the right.The system on the left (SOL) is composed of stages from *start* to the enqueuing stage of the queue. The system on the right (SOR) is composed of stages from the dequeuing stage of the queue to the *sink*. Both of these systems are taken into consideration when balancing this queue. The objective here is to remove the imbalances in the processing rates of the systems on both sides of the queue. The algorithm takes different steps to remove these imbalances depending on whether the queue is empty or full.

Queue is full A full queue means that the processing rate of the SOL is greater than that of the SOR. Some tasks have to be moved from SOL to SOR. The number of tasks moved depends on the difference in the processing rates of SOL and SOR. After the tasks are moved, the output of SOR should become greater than that of SOL, i.e., the rate of item-processing of

the bottleneck stage of the SOR should become greater than that of SOL.

Queue is empty An empty queue means that the processing rate of SOR is greater than that of the SOL. Tasks have to be moved from SOR to SOL until the processing rate of SOL becomes greater than that of the SOR. Thus the number of tasks moved depends on the relative processing rates of SOR and SOL. After the tasks are moved, the processing rate of SOL should become greater than that of SOL.

Some stages make use of special resources of the machines on which their tasks execute (tape drives, for example). These stages are marked *unbalancable*, and are not touched by the Load Balancer. In addition, the program for a stage might be available only for certain of the machines on the network, if the network is heterogeneous. The Load Balancer will only assign a processor to a stage if the processor can execute a program for the stage.

At regular intervals, the load balancer collects event processing rates from all the tasks. The duration of the intervals varies depending on the event processing rate of the system. The minimum limit on the length of the interval is 30 seconds. The Load balancer needs all the tasks to process at least ten events before it can determine their item processing rates accurately. The Load balancer starts from the minimum interval and doubles the interval length till all tasks can process at least ten events.

There are two important implementational details. THe first detail is detecting when a queue is full or empty. Our implementation considers a queue to be full if its average length over the past interval is greater than 90% of its capacity and empty if its average length is less than 10% of its capacity. This permits a rapid detection of an imbalance.

The second detail is to select which full or empty queue shoudl be selected for balancing first. In general, more than one queue will be full or empty, because the bottleneck stage will back up the system on the left and starve the system on the right. The queeu closest to the sink is considered first. If the queue is full, it is chosen. If the queue is empty, the bottleneck stage might still be to the right. So, the algorithms searches to the left for the last empty queue and chooses ir for balancing.

4 Evaluation

We will evaluate the Load Balancer in this section and present the results. We conducted two experiments. In the first experiment we show that dynamic monitoring and balancing reduces the job completion time compared to if the job is run with just the static re-allocation. In the second experiment we test the Load Balancer with a system in which the event processing rates of the tasks change during the execution

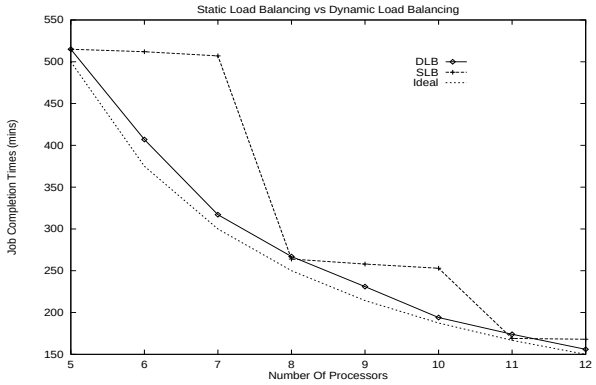

Figure 3: Static load balancing vs dynamic load balancing

of the job. We executed the experiments on a network of DEC Alpha workstations connected with FDDI and running UFMulti.

4.1 Synthetic Job

The synthetic job consists of five stages named Start, Middle1, Middle2, Middle3, and End. Netqueues Q1, Q2, Q3 and Q4 connect these stages in a manner similar to that shown in Figure 1. The Start job creates an event and puts it into Q1 once every Sy seconds. Each job Middle$i$ repeatedly dequeues an event, spends $M_i x$ seconds processing the event, puts the processed event into the next queue, then performs bookkeeping for $M_i y$ seconds. The END job spends Ex processing each event. The synthetic job processed 10,000 events, and each event description required 1012 bytes. The START and END stages are unbalancable, the Middle$i$ stages are balancable.

4.2 Static LB vs Dynamic LB

The following experiment compares job completion times of the static, dynamic and ideal LB algorithms. Static LB (SLB) performs the static re-allocation and does nothing else. Dynamic LB (DLB) does static re-allocation and also dynamic monitoring and balancing. Ideal LB (ILB) is similar to the static LB except that the processors can be allocated in fractions. We use ILB to give a lower bound on the completion times.

The synthetic job was set up so that the Start and End tasks have no delay in item processing, and the Middle$i$ tasks use three seconds per event. This job is run multiple times using between 5 and 12 processors. Each time the job is run with both static LB and dynamic LB. The results are shown in Figure 3, along with the results of ideal LB.

When the number of processors is five, each stage of the job gets one processor and there are no extra processors to balance. Consequently, SLB and DLB result in the same completion time. Ideal LB results in a smaller completion time as it is not the result of the

experiment, but computed using the item processing rates. It does not take into account the start up time of the job and the network delays.

With six or seven processors, SLB assigns only one processor to one of the Middle*i* stages and achieves no performance improvement. DLB achieves a significant improvement with the additional processors, by performing a fractional allocation of the extra processor to the three stages. After the initial reallocation, the system will be unbalanced. The Load Balancer will detect the imbalance and reshuffle the processor allocation. While the system will still be imbalanced, a great deal of work can get done before queues again become full or empty, due to the buffering in the Netqueues. This phenomena repeats when we apply 8 and 9 processors to the system. THe performance of BLD is close to that of ILB, indicating that the virtual fractional proessor allocation is working well.

4.3 Phased Jobs

We want to determine if DLB reacts well to changes in the event processing rates. In the previous experiment, the item processing rates of the tasks remained unchanged throughout the run time of the job. As we discussed, the stages of a job will often experience a varying event processing rate. To test this scenario, we designed an experiment in which the event processing rate changes.

We divided the job into phases. At the beginning of each phase, each Middle*i* stage selects a new event processing time uniformly randomly between 2 and 12 (we stored these samples in a configuration file to ensure that all experiments gave comparable results). We fixed the number of processors at 10 for all experiments, and varied teh number of phases between 2 and 110.

In Figure 4, we plot the performance of the SLB and the DLB algorithms. We plot two the performance of two comparison algorithms, Ideal Load Balancing (ILB) and Phase-wise Static Load Balancing. For ILB, the optimal fractional allocation is recomputed at the beginning of every phase and for PSLB the optimal integral allocation is recalculated at the beginning of every stage.

When the number of phases is small, the completion times with DLB are much smaller than that of SLB or PSLB and very close to the completion times of ideal LB. As the number of phases increases, the item processing rates of the stages change frequently and the performance of the DLB decreases. Its completion times increase and are close to those of PSLB. However, the performance of DLB is always between that of PSLB and DLB. Note that PSLB is theoretical and ew plot a calculated result for it, while SLB and DLB are implemented and we plot experimental measurements for them.

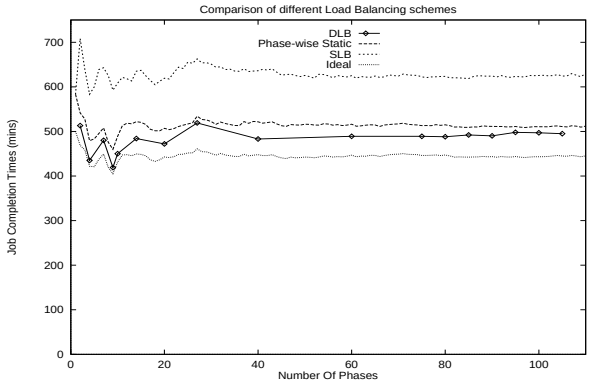

Figure 4: Comparison of LB algorithms with phased jobs.

5 Conclusions

We present our results on load balancing in UFMulti, a package for High Energy Physics computing on a distributed system. The nature of HEP computations leads one to structure the computation as a chain of processing stages, with each stage connected by a Netqueue. The bottleneck stage in the chain limits the throughput of the system. We take advantage of the computation structure to perform simple but very effective load balancing. By making use of the buffering in Netqueues, our dynamic load balancing algorithm can make a virtual fractional allocation of]a procesor to a processing stage. We present experimental data that shows that our algorithms are very effective.

Future Work

We plan to extend our load balancer by applying it to systems that are DAGS. Such an extention requires knowledge of the rate that different path are taken, and a more sophisticated bottleneck detector. We also plan a tighter integration with the Resource Monitor to choose lightly loaded hosts to be balancing processors.

References

[1] *UFMulti Distributed Toolkit*, http://www.phys.ufl.edu/∼ufm, 1994

[2] P. Avery et al. The UFMulti Project, *International Conference on Computing in High Energy Physics*, 1994 pg. 156 - 164

[3] Aric F.Zion. *NetQueues: A Distributed Processing System For High-Energy Physics* Masters Thesis, Dept. of CIS, University of Florida, 1992

[4] Y.T. Wang and R.J.T. Morris. Load Sharing in Distributed Systems *IEEE Transactions on Computers,* C-34(3), March 1985, pp. 204-17

[5] T.L. Casavant and J.G. Kuhl. A Taxonomy of Scheduling in General Purpose Distributed Computing Systems *IEEE Transactions on Software Engineering,* SE-14(2), February 1988, pp. 141-54