The EASY - LoadLeveler API Project

Joseph Skovira, Waiman Chan, Honbo Zhou
International Business Machines
David Lifka
Cornell Theory Center

Abstract. *With the increasing use of distributed memory massively parallel machines (MPPs) such as the IBM SP, the need for improved parallel job scheduling tools has sparked many recent developments. IBM's LoadLeveler is being used at the Cornell Theory Center, but problems exist with the current scheduling algorithm applied to the job mix on the 512-node SP. In order to address Cornell's difficulties, Joseph Skovira began to consider enhancements to LoadLeveler. At about the same time, David Lifka, developer of the EASY parallel job scheduler, began working at CTC. With Waiman Chan and Honbo Zhou of IBM LoadLeveler development, we have developed a LoadLeveler API that allows external schedulers like EASY to control the starting and stopping of jobs through LoadLeveler.*

1 Introduction

The EASY-LoadLeveler collaboration is interesting for several reasons. First, both EASY and LoadLeveler benefit. By interfacing to LoadLeveler, the EASY algorithm can more readily be expanded to heterogeneous MPPs using information already maintained by LoadLeveler. LoadLeveler benefits because parallel job scheduling can be externalized where it can then be easily customized for requirements at different sites. Second, both LoadLeveler and EASY have been proven on MPPs – both codes scale and have a measure of stability because of their maturity. Finally, this work introduces a new model for job scheduling. In effect, the task is segmented into administration and scheduling. In our case, LoadLeveler handles the administration (e.g., the job queue) information regarding node resources of the machine, status of jobs running throughout the machine, and information about the machine node status. EASY takes care of the job scheduling, that is, when a particular job should run and on which nodes it should run. Since the interface is externalized to a small number of calls, and since EASY is written in Perl, changes to the scheduling algorithm or policy can be quickly made at different sites without affecting the compiled code of LoadLeveler.

2 Background of EASY and LoadLeveler

EASY was originally developed at Argonne National Laboratory for use on Argonne's 128-node SP system. As required by many sites, Argonne had parallel

job scheduling requirements not met by any commercial or research project then available (including LoadLeveler). The EASY algorithm was developed with several fundamental principles in mind. First, the job queue would be ordered by submission time – jobs would be considered for execution in the order they were submitted. If the first job in the queue could be scheduled to run on available nodes, it would be executed. If the next job in the queue could not run because enough nodes were not available, it would wait until enough nodes became free in order to run. This leads to the second principle: the scheduling algorithm is deterministic. Jobs in the queue are never delayed from running by jobs submitted to the queue after them. Finally, while a job is waiting for resources to free, smaller jobs further down the queue can be run as long as they complete before the waiting job is scheduled to run. In this way, jobs can be backfilled onto available nodes in order to make better use of machine resources. EASY is currently used by many MPP sites throughout the world and is known for its efficient scheduling, simplicity, and robustness. One note is that EASY schedules homogeneous nodes of an MPP. A mechanism in the algorithm to deterministically schedule special resources is currently under development.

Referring to the definition in [1], Loadleveler is a distributed, network-based, job scheduling program. LoadLeveler is a modified version of Condor that is sold by IBM. Condor was originally designed to schedule clusters of workstations. LoadLeveler was built on this base as a scheduler for MPP's, in particular, the IBM SP. LoadLeveler will locate, allocate, and deliver resources from across a network while attempting to maintain a balanced load, fair scheduling, and an optimal use of resources. The goal of LoadLeveler is to make better use of existing resources by using idle compute nodes and to optimize batch throughput. LoadLeveler is also used by many SP sites worldwide and is a product fully supported by IBM. However, the code is written in C and changes to its function (including parallel job scheduling) require redesign and re-release of the product by IBM.

3 Combining EASY and LoadLeveler

Although EASY is a standalone scheduling system that is proven to work on existing MPP systems, there are several reasons to integrate EASY with LoadLeveler instead of just enhancing EASY. First, LoadLeveler contains much more configuration information regarding the machine than EASY records, especially regarding differences between the nodes of a machine. The machine information is updated using the SP resource manager so that system changes are quickly reflected in the LoadLeveler data. Because these records and distribution mechanisms have proven reliable in LoadLeveler, we decided to take advantage of them in order to enhance EASY. Future versions of EASY will make use of this information to perform exact resource matching.

Another advantage occurs because LoadLeveler maintains daemons at every SP node. These Startd daemons are used to start user tasks at a particular node of the machine. Because these daemons are in place, starting and stopping jobs by the EASY scheduler occurs much faster than with standalone EASY, which uses rsh. This eliminates some amount of startup and shutdown overhead.

Using the new system, it becomes much simpler to switch between LoadLeveler and EASY scheduling. In fact, future modifications might include the capability to partition a single machine into pools of nodes which are scheduled using 2 (or more) different algorithms. This might prove useful as a way to take advantage of different algorithms strengths depending on the parallel job mix.

At the start of this effort, we considered which scheduling algorithm to interface with LoadLeveler, and the choice of EASY was a clear one. EASY is currently one of the most popular schedulers available for parallel jobs that provides deterministic queuing combined with straightforward interface features and simple administration. Also, being written in Perl, the algorithm is simple to understand and modify (both in terms of scheduling and policy). Both EASY and LoadLeveler are supported; LoadLeveler as an IBM product and EASY as a public domain code maintained by David Lifka. To cap the decision, the LoadLeveler group realizes the worth of the EASY algorithm, and David Lifka understands the information from LoadLeveler that can be used to enhance the EASY algorithm for resource scheduling.

A final choice to make was whether to include the EASY algorithm within LoadLeveler or to provide an interface for use with a modified version of the EASY code. The choice of an API was clear for several reasons. Including the EASY algorithm within LoadLeveler would require a rewrite from Perl to C, which would add time to the projected deployment of this solution. In fact, inclusion as C code proves more damaging due to the more static nature of the result. In order to change the scheduling algorithm, there would have to be a release of the LoadLeveler product. Even then, the scheduling algorithm would be static for all sites. By using an API, changes can be both prototyped and deployed quickly. In addition, different sites can implement different scheduling solutions tailored to their individual job mixes – a very powerful aspect of this solution. In fact, the API allows the EASY code to be totally replaced by an algorithm entirely designed by another SP administrator. As long as the API calling formats are followed, any scheduling algorithm can be implemented. The API may also be used to develop system administration tools an example of which is the service policy currently implemented by EASY. Using an external tool, nodes could be taken out of service quickly without restarting and reconfiguring LoadLeveler. Because the schedule API provides information about the nodes and the jobs, any reconfiguration of these data structures is possible by an external tool.

4 The Application Programming Interface between LoadLeveler and EASY

From the outset, we intended the API to be as straightforward as possible. Our first version was intended to provide the minimum functionality necessary to interface the EASY scheduler. As we learn more about the functioning of external scheduling algorithms, the interface will be appropriately expanded. The current interface consists of 4 calls. These calls are illustrated in figure 1, which also includes an illustration of job division between LoadLeveler and EASY. In figure 1, note that there were 3 tasks required to interface LoadLeveler with EASY. The first was to enhance LoadLeveler to provide the appropriate calls to be used by EASY. Next, the standalone EASY code was modified to interface with the new API functions. Finally, an interface layer was developed to convert the information returned in the LoadLeveler C structures into data as required by the EASY code, written in Perl. The following functions are provided by LoadLeveler to support external scheduling using the EASY scheduler:

`ll_get_job`: Retrieve the job queue - This call returns the contents of the LoadLeveler job queue. All jobs currently in the queue, including those running and waiting to run, are returned by this call.

`ll_get_node`: Retrieve the node status - The status of all SP2 nodes in the system is returned by this call.

`ll_start_job`: Start a job - EASY uses this call to begin the execution of a previously queued parallel job. Once EASY has determined that a job is to be started, it issues this call to LoadLeveler, along with a specific set of nodes on which to start the parallel job.

`ll_cancel_job`: Cancel a job - EASY issues this command to cancel a job when the execution time has exceeded a limit which the user has set.

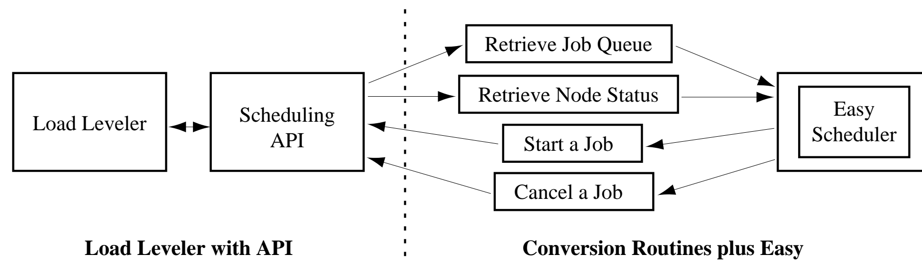

Figure 1 - Job partitioning and interface calls

Figure 2 shows how EASY would make use of the provided calls to examine the current job queue and machine status, decide which job to schedule, then start a job. At Cornell, this new version is being tested, maintaining the existing LoadLeveler interfaces for the users (which include job submission and job

command files). In addition, this new version will provide a wall-clock run-time variable in LoadLeveler, which EASY requires, and EASY commands for examining the operation of the job scheduling system. EASY tools like spq, spusage, and xspusage will become available for the Cornell users.

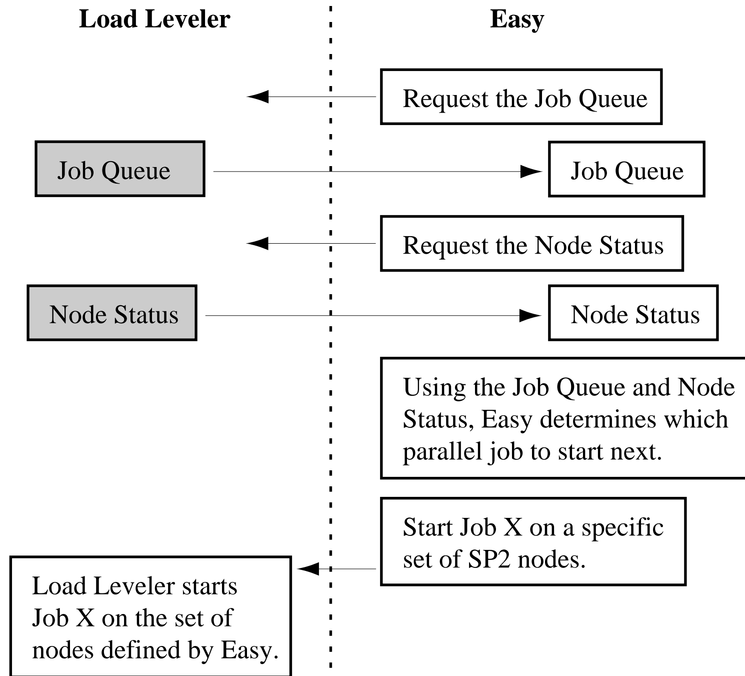

Figure 2 - Starting a job across the Interface

5 Performance and Scalability

During our initial testing, we have started to collect performance statistics for the LL-EASY code. Figures 3 and 4 shows the results obtained from submitting 1000 jobs to an 8 node SP2 system using LL-EASY. Figure 3 shows the time it takes to start jobs from the FIFO queue. These jobs were not backfilled into available resources but, rather, started once they arrived at the top of the queue. The startup time axis represents the total startup time beginning with the scheduler requesting job information from LoadLeveler, deciding which job to start, calling LoadLeveler to start the job, and completing with the job actually being dispatched by LoadLeveler to the appropriate nodes. The longest time recorded was 13 seconds for job 31, the average was approximately 5 seconds. Note that the time decreases as the job queue is consumed (as expected). The discontinuities in the graph are caused by a lock on a resource file maintained by EASY which is asynchronously accessed by multiple daemons. We are developing a fix to solve this issue, but, even with this effect, jobs start very quickly. There is

very little overhead in transferring even a large job queue from LL to EASY. Note also that rapid startup of the jobs is further enabled by the existence of the LL daemons at each of the nodes.

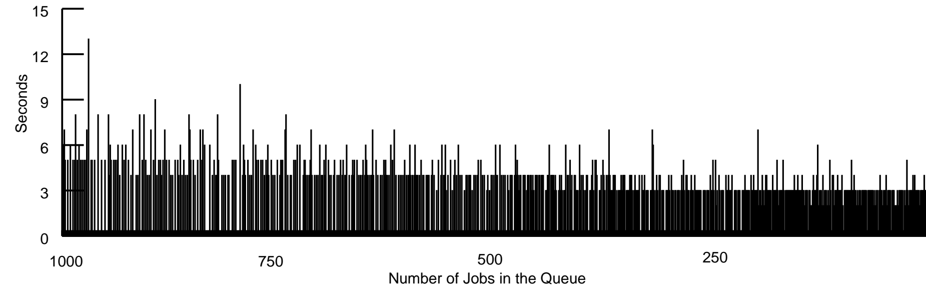

Figure 3 - FIFO Jobs

Figure 4 also shows job number versus submission time, but in this case, the jobs started were backfilled. These jobs were started as the result of a search of the queue to attempt to fill vacant resources while waiting to start the top job in the queue. Note again that the worst case time is 12 seconds for job 220 and that the average is approximately 5 seconds. Also note that the number of backfilled jobs reduces near the end of the test, also as expected. This occurs because the smaller, shorter running jobs are used early to fill the holes in the schedule. As time passes, there are fewer of these small jobs which can be used to fill available nodes, consequently, fewer jobs are backfilled later in time. This effect disappears with an infinite queue (the equivalent of many users!) since smaller jobs are always available to use the idle nodes. Again, the discontinuity of the data is partly due to accessing the resource file shared by the daemons. However, some of the spike amplitude in the backfill data is due to searching deep into the queue to find a job to backfill (note the spike for job 730). Although the search is quickly performed, it is not done for the FIFO jobs of figure 3, so later spikes in the backfill data of figure 4 tend to be larger than those of figure 3. Nonetheless, the time to start backfill jobs is also very short.

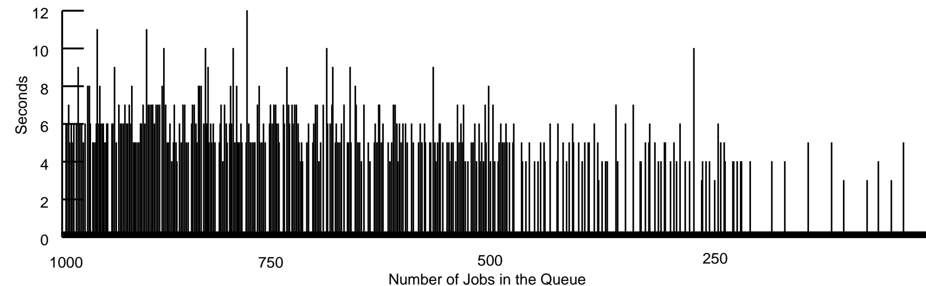

Figure 4 - Backfilled Jobs

So far, performance data for the algorithm is excellent. The short startup times indicate that the combined LL and EASY algorithms operate efficiently and can

dispatch jobs even for a large queue in under 13 seconds per job, worst case, and under 5 seconds per job, on average. With this low overhead, utilization of the machine is not hindered by the scheduling algorithm.

6 Future Plans

We plan to incorporate deterministic scheduling of heterogeneous resources in the EASY scheduling algorithm by Spring 1996. This will include enhancements to the LoadLeveler API so that resource information can be passed to the scheduler. We also plan to incorporate dynamic process allocation in such a way that does not break the determinism of the EASY algorithm. EASY keeps track of the number of nodes that are available and that will not be required for the jobs waiting in the queue. If a parallel job needs additional resources during a run, a mechanism will be provided for the job to ask EASY for access to these nodes. We plan to build this capability in by late 1996.

References

1 Kaplan, J., Nelson, M., *A Comparison of Queueing, Cluster and Distributed Computing Systems.* NASA TM 109025 (Revision 1) - 1, NASA Langly Research Center, (1994)

2 Lifka, D., Henderson, M.,and Rayl, K.,ANL/MCS-TM-201, *Users Guide to the Argonne SP Scheduling System*, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL (1995)

3 Lifka, D., *The ANL/IBM SP Scheduling System*, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL (1995)

4 Rosenkrantz, M., Schneider, D., Leibensperger, R., Shore, M., Zollweg, J., *Requirements of the Cornell Theory Center for Resource Management and Process Scheduling*, Cornell Theory Center, Ithaca NY (1995)

This article was processed using the LaTeX macro package with LLNCS style