The UnderLord Scheduler

A Drop-In Scheduler for the Portable Batch System

Walt Akers, Chip Watson, Jie Chen, Ying Chen

June 8, 2001

TJNAF - Thomas Jefferson National Accelerator Facility

DOCUMENT DATE:

Table of Contents generated: June 8, 2001 3:19 pm

TRADEMARKS:

UNIX is a registered trademark of AT&T in the USA and other countries.
Linux and gmake are a register trademark of the Free Software Foundation.
The X Window System is a trademark of Massachusetts Institute of Technology.
OSF / Motif and Motif are trademarks of Open Software Foundation, Inc.
PBS and Open PBS is a trademark of Veridian Systems.

SURA / TJNAF:

The Southeastern Universities Research Association (SURA) operates the Thomas Jefferson National Accelerator Facility (TJNAF) for the United States Department of Energy under contract DE-AC05-84ER40150.

i. Table of Contents

ii. List of Figures

1. Introduction

Project Overview

The UnderLord scheduler was developed at Jefferson Lab as part of a joint effort with MIT to develop a meta-facility between our institutions. The implementation of this meta-facility requires two scheduling systems: the OverLord, a meta-scheduler responsible for migrating jobs between sites; and the UnderLord which is responsible for deploying jobs locally. The UnderLord scheduler represents the first step of this larger project.

To promote code reuse and modularity, we have developed the UnderLord as a collection of C++ classes. The bulk of this work resides in two libraries:

PBS Server Library — A collection of classes that provide simplified access to the data stored within the PBS Server and the Machine Oriented Mini-Servers (PBS MOM).

UnderLord Library — A collection of extensible classes that provide mechanisms to order job candidates and select the best time and location for them to be executed.

Document Objectives

Since this project is under ongoing development, this version of the documentation will be limited to the following issues:

- Provide instructions on how to build, install and start the UnderLord scheduler.
- Describe the structure and syntax of the configuration file that guides the scheduler's decisions.
- Describe the different options that can be set in within the scheduler’s configuration file.
- Define the mechanisms that the UnderLord uses to sort jobs.
- Demonstrate the technique for extending the sorting mechanism to support site specific objectives.
- Explain the general match-making approach used by the UnderLord to get optimal system utilization.
- Discuss future objectives for the ongoing development of the project.

Tested Systems

As of this writing, the UnderLord scheduler has been built and tested on RedHat Linux version 6.2 through 7.1 for both Alpha and Intel processors. Although we have not tested on other platforms, we believe that this product should be readily portable to all of the systems supported by the Portable Batch System.

2. Building, Installing and Starting the UnderLord

Overview

While the UnderLord was designed to be an extensible framework, it is distributed in a *readily usable* default configuration. The term 'readily usable', of course, only applies if your scheduling objectives are similar to ours.

This section will describe how to obtain, configure, build, install and start the UnderLord scheduler with the default configuration on a Linux system.

Obtaining the Distribution

The source code and documentation for the scheduler can be obtained as a zipped tar file from the High Performance Computing site at Jefferson Lab:

http://www.jlab.org/hpc/UnderLord/underlord_1.0.2.tar.gz

Prepare the Compilation Environment

Once you have downloaded and extracted the source distribution you are ready to begin compilation. The PBS source distribution should be installed and the following environment variables should be defined.

PBS_SRC — This variable should point to the root of the PBS source tree.

At our site we create a link to the most recent version of the PBS source distribution on the /usr/local/PBS_SRC directory.

PBS_HOME — This variable should point to the root of the PBS binary installation. The bin, sbin, and lib directories for PBS will be located here.

At our site we create a link to the most recent PBS binary tree on the /usr/local/PBS_HOME directory.

Update PBS Header Files

Since PBS was developed as a system of C libraries, the header files contain several inconsistencies that prevent the headers from being included directly into C++ source files. Prior to compiling the UnderLord it will be necessary to correct these problems.

- On lines *105* and *107* of the header file *log.h*, the C++ keyword “class” is used as a variable name. To correct this change “class” to “Class”.
- On line *191* of the header file *pbs_error.h*, the function prototype for *pbse_to_text* requires an integer argument. To correct this change:

  *extern char * pbse_to_text (); to*

  *extern char * pbse_to_text (int);*

As of version 2.3.12 of Open PBS, these are the only inconsistencies that need to be corrected.

Build and Install the UnderLord

With the environment variables set and these corrections made, you are ready to build the UnderLord scheduler. Change directories to the UnderLord source directory and run *gmake* to compile the collection of libraries, applications and test programs.

Note that in order to make errors and warnings easier to read, much of the output from these makefiles has been suppressed. The distribution should compile without any errors or warnings. Once the build is complete run *gmake install* to install the binaries.

Note: Because of the way that the UnderLord scheduler is designed, it is necessary to build, install and execute it on the same host where your PBS Server is running. This is because the UnderLord requires direct read-access to the server’s accounting files in order to track current and historic system utilization.

3. The UnderLord Configuration File

Configuration File Location

The UnderLord configuration file is located in the */usr/spool/PBS* directory and is named *scheduler.cfg*. This file contains the configuration options that define how the scheduler will prioritize jobs and how they will be deployed on the system.

You should note that the UnderLord comes with a pre-installed configuration file that is copied to the /usr/spool/PBS directory every time the scheduler is rebuilt/reinstalled. If you plan to rebuild the scheduler, it is important to safeguard the site specific scheduler.cfg prior to reinstalling... otherwise, it will be overwritten.

Configuration File Format

The format of this file is very similar to the "*.ini file*" format used in early versions of Microsoft Windows. At the most fundamental level, the configuration file contains three types of elements: sections, entries, and comments.

Section Headers — *example: [Section Header]*

A section header is a character string enclosed in square brackets that defines the beginning of a group of related entries. Sections that are predefined in the default scheduler configuration file include:

[Scheduler]

[Job Age Stage]

[Fair Share]

[Restricted Nodes]

Once a section has been started, all entries that follow it are considered to be part of that section until the next section begins *or* the file ends.

Entries — *example: entry=value*

An entry consists of a character string *tag* and an associated character string *value*. Entries ***must be*** defined within the confines of a section.

Comments — *example: #comment*

Comments are preceded by a '#' character and continue until the next newline character.

The following is a brief example showing the structure and format of the scheduler configuration file.

Figure 1: Sample UnderLord Configuration File

```
# This section defines entries associated with the scheduler
[Scheduler]
summaryRate=10:00
summaryDir=/usr/spool/PBS/sched_logs

# This section defines entries associated with Fair Share
[Fair Share]
default=1
```

4. Configuration Options for the UnderLord

[Scheduler] Options

The Scheduler section of the configuration file contains entries that effect the behavior of the UnderLord scheduler as a whole. This section has the following entries.

summaryRate — *summaryRate=10:00*

The summaryRate entry specifies how frequently a job summary will be written to disk. The value is provided in hour:min:sec format. Whenever the summaryRate period expires, the UnderLord will open a summary file in the configuration specified *summaryDir* and will write a list of waiting jobs and when they are projected to start and finish.

The default value is 10:00 (ten minutes).

summaryDir — *summaryDir=/usr/spool/PBS/sched_logs*

The summaryDir entry specifies the directory where summary files will be written. The directory specified in this entry is used by both the *stageSummary* and the *execSummary* options.

The default directory is /usr/spool/PBS/sched_logs

stageSummary — *stageSummary=yes*

This is a yes/no entry that will tell the UnderLord whether it should write a sorting summary report to an output file in the *summaryDir*. If this flag is set to "yes", then whenever the summary period expires the UnderLord will write a summary describing the current conditions (utilization, priority, etc) under which sorting decisions are being made. This is a useful tool for fine-tuning the parameters used by the various sort stages.

The text file that is generated by this option is recreated every 24 hours and has a date oriented name: *yyyymmdd.sorting*.

execSummary — *execSummary=yes*

This is a yes/no entry that will tell the UnderLord whether it should periodically report on projected start/completion times for jobs in the queue. This report can be used to determine when jobs are actually projected to be run.

The text file that is generated by this option is recreated every 24 hours and has a date oriented name: *yyyymmdd.summary*.

accountingDir — *accountingDir=/usr/spool/PBS/server_priv/accounting*

This is the directory where the server's ACCOUNTING logs are located. As you'll note, the scheduler's need to access the server's accounting logs in the *server_priv* directory is one of the reasons that it must be executed on the same host as the PBS server.

maxHistory — *maxHistory=14*

This is the number of days of history that should be initially retrieved from the server log database.

maxWallHours *maxWallHours=200*

Once a job in the accounting database accrues this much wall time without having an end event in the log, it is assumed to be a lost job. The job will be eliminated from the internal database to prevent it from skewing the weight computations.

[Job Duration Stage] Options

Since the UnderLord is a multi-stage scheduler, each scheduling stage has specific configuration options that define its behavior. This section contains entries that define the characteristics of the Job Duration Stage of the scheduling process.

weight *weight=1*

The weight entry defines the significance of the Job Duration Stage in relationship to all other stages in the scheduling process. If this value is 0, then the stage will have no impact on job scheduling.

The default value is 1.0.

lowerBound *lowerBound=1:00*

Since all jobs are fitted to a curve, the lowerBound identifies the duration (in seconds) that corresponds to the bottom 5% of the curve. All values less than the lowerBound will be fitted asymptotically toward 0. Those values between the lowerBound and upperBound will be fitted linearly between the two values.

The default value is 1:00 (1 minute).

upperBound *upperBound=24:00:00*

The upperBound identifies the durations (in seconds) that corresponds to the upper 95% of the curve. All values greater than the upperBound will be fitted asymptotically toward 1.

The default value is 24:00:00 (1 day).

countProcs *countProcs=yes*

This is a yes/no entry that defines whether the Job Duration Stage should multiply the job's duration by the number of processors requested.

The default value is yes.

[Job Age Stage] Options

This section contains entries that define the characteristics of the Job Age Stage of the scheduling process.

weight *weight=1*

The weight entry defines the significance of the Job Age Stage in relationship to all other stages in the scheduling process. If this value is 0, then the stage will have no impact on job scheduling.

The default value is 1.0.

lowerBound *lowerBound=1:00*

Since all jobs are fitted to a curve, the lowerBound identifies the time in queue (in seconds) that corresponds to the bottom 5% of the curve. All values less than the lowerBound will be fitted asymptotically toward 0. Those values between the lowerBound and upperBound will be fitted linearly between the two values.

The default value is 1:00 (1 minute).

upperBound *upperBound=24:00:00*

The upperBound identifies the time in queue (in seconds) that corresponds to the upper 95% of the curve. All values greater than the upperBound will be fitted asymptotically toward 1.

The default value is 24:00:00 (1 day).

[User Priority Stage] Options

This section contains entries that define the characteristics of the User Priority Stage of the scheduling process.

Note: User priority is the priority value that the user applies to each job that he or she submits. This value and this stage does not influence the scheduling of jobs submitted by different users.

weight *weight=1*

The weight entry defines the significance of the User Priority Stage in relationship to all other stages in the scheduling process. If this value is 0, then the stage will have no impact on job scheduling.

The default value is 1.0.

[Queue Priority Stage] Options

This section contains entries that define the characteristics of the Queue Priority Stage of the scheduling process. Using PBS, each queue is granted an integer priority. This stage considers that priority to represent the queue's fair share of system resources. Using historic and current data it will increase or decrease the queue's weight depending on the amount of system resources that its's jobs have consumed. Considered resources include walltime, cpu time, and number of jobs executed. The system administrator may change the significance of any of these factors by altering the appropriate parameters in this section.

weight *weight=1*

The weight entry defines the significance of the Queue Priority Stage in relationship to all other stages in the scheduling process. If this value is 0, then the stage will have no impact on job scheduling.

The default value is 1.0.

wallTimeFactor *wallTimeFactor=1.0*

The significance that the queue's accumulated walltime will have in the formula used to compute the queue's priority.

The default value is 1.0.

cpuTimeFactor *cpuTimeFactor=1.0*

The significance that the queue's accumulated CPU time will have in the formula used to compute priority.

The default value is 1.0.

runJobFactor *runJobFactor=1.0*

The significance that the queue's accumulated number of running jobs will have in the formula used to compute priority.

The default value is 1.0.

recentJobThreshold *recentJobThreshold=24:00:00*

A time value given in hh:mm:ss format. This value is the amount of time following job completion that a job is considered recent.

Typically one day (24:00:00).

oldJobThreshold *oldJobThreshold=168:00:00*

A time value given in hh:mm:ss format. Once the time following a jobs completion is greater than this value, that jobs statistics will no longer be considered in calculating the queue's priority.

Note: The PBS_ServerDatabase class purges job records after two weeks. Using an oldJobThreshold longer than two weeks will be ineffective without first modifying the PBS_ServerDatabase class.

Typically one week (168:00:00)

oldFactor *oldFactor=0.10*

The multiplier that will be used to decrease the significance of old jobs *(jobs that have ended between the oldJobThreshold and the recentJobThreshold)*.

By default this value is 0.10.

[User Share Stage] Options

This section contains entries that define the characteristics of the User Share Stage of the scheduling process. This stage is very similar to the Queue Priority stage. It considers the user share value to represent the user's fair share of system resources. Using historic and current data it will increase or decrease the user's weight depending on the amount of system resources that his jobs have consumed. Considered resources include walltime, cpu time, and number of jobs executed. The system administrator may change the significance of any of these factors by altering the appropriate parameters in this section.

Note: Each user's share of system resources is specified in the Fair Share section.

weight *weight=1*

The weight entry defines the significance of the User Share Stage in relationship to all other stages in the scheduling process. If this value is 0, then the stage will have no impact on job scheduling.

The default value is 1.0.

wallTimeFactor *wallTimeFactor=1.0*

The significance that the user's accumulated walltime will have in the formula used to compute the his fair share of resources.

The default value is 1.0.

cpuTimeFactor *cpuTimeFactor=1.0*

The significance that the users's accumulated CPU time will have in the formula used to compute the his fair share of resources.

The default value is 1.0.

runJobFactor — *runJobFactor=1.0*

The significance that the user's accumulated number of running jobs will have in the formula used to compute his fair share of resources.

The default value is 1.0.

recentJobThreshold — *recentJobThreshold=24:00:00*

A time value given in hh:mm:ss format. This value is the amount of time following job completion that a job is considered recent.

Typically one day (24:00:00).

oldJobThreshold — *oldJobThreshold=168:00:00*

A time value given in hh:mm:ss format. Once the time following a jobs completion is greater than this value, that jobs statistics will no longer be considered in calculating the user's fair share of resources.

Note: The PBS_ServerDatabase class purges job records after two weeks. Using an oldJobThreshold longer than two weeks will be ineffective without first modifying the PBS_ServerDatabase class.

Typically one week (168:00:00)

oldFactor — *oldFactor=0.10*

The multiplier that will be used to decrease the significance of old jobs (jobs *that have ended between the oldJobThreshold and the recentJobThreshold)*.

By default this value is 0.10.

[Fair Share] Options

This section contains entries that define each user's fair share of system resources. The default entry will be applied to any user who does not have an entry in this section. In the event that the user does not have an entry and there is no default entry defined, then the user will arbitrarily be assigned a share of 1.0.

default — *default=1.0*

This is the default share of system resources that will be applied to any user who does not have a specifically assigned share.

[Restricted Nodes] Options

One of the most notable features of the UnderLord scheduler is it's ability to associate a list of nodes with a specific queue. If the system administrator wants to bind a node to one or more specific queues, he should make an entry here with the name of the node followed by the queues that are authorized to use it. The entries in this section are of the form:

node — *node=queue1,queue2*

The word 'node' is not an entry keyword in this section. 'node' should be replaced with the name of the node that the queue should be associated with.

Note: If a queue is not mention in this section, then it is assumed to have access to all available nodes. Also, if a queue is identified for a node, then all of the other nodes that are associated with it must also be listed as line entries.

In this example, the queue Slave is only allowed to deploy jobs against node hpc14b.

```
[Restricted Nodes]
hpc14b=Slave
```

5. How the UnderLord Sorts Jobs

General Scheduler Requirements

In general, a PBS scheduler is required to evaluate jobs that are waiting in one or more queues of a PBS Server and determine which of these jobs should be executed next. The scheduler may consider any number of parameters in making this decision, but, in the end, the aggregate job selection over time should conform to a collection of scheduling policies that have been determined by the site.

A generic site might have a policy stating that, over a period of time, all user's should have an equal share of system resources. Other sites might stipulate that resources be divided evenly (or unevenly) based on the queue from which a job originates.

While it is not the job of the scheduler to dictate policy, it should have the capabilities to implement whatever reasonable scheduling policy the site may implement. Within the UnderLord scheduler, we do this by using a multi-stage weighting algorithm.

Multi-Stage Job Weighting

Early on in our design, we realized that there were many factors for the scheduler to consider during job sorting. Jobs might be sorted based on their time in queue, projected duration, number of nodes requested, the queue or user's past utilization of resources or many other factors. Unfortunately, it is difficult to prioritize this list of factors because the importance of any single statistic might change based on the current job mix.

For instance: If we developed a scheduling algorithm that considered each job's *projected duration* and it's *time in queue* (giving priority to the duration factor), then shorter jobs would always run first. The *time in queue* factor would only be considered when two competing jobs had the same *projected duration*. In the long term this would be brutally unfair to a job that is only nominally longer than it's competitors. In fact, the longer job might wait in the queue indefinitely.

Consequently, our example algorithm must have the intelligence to consider both *duration* and *time in queue* simultaneously when making scheduling decisions. It must determine when a generally less significant factor (such as time in queue) has reached sufficient magnitude to become the driving term. While this approach is relatively simple when dealing with two factors, it becomes more difficult as more sort terms are added.

To address this issue, the UnderLord uses a series of sorting stage objects (derived from the C++ UnderLordStage class) that examine a specific sort factor (such as duration) and assign each job a weight that varies between 0 and 1.0 --- with 1.0 being the most significant.

Bounding the Curve

Having an infinite number of values between 0 and 1.0, it is necessary for the sorting stage to fit its values into that range such that a statistically outlying value does not overwhelm the significance of the other values. To solve this problem, the UnderLord uses a curve fitting scheme where an upper and lower boundary of significance is specified and these values are used to bound the curve.

Computing the Raw Weight

It is assumed that each job has a series of parameters that can be used for sorting. Conceivably, these numeric parameters may fall anywhere between negative and positive infinity. To most effectively use these values it is necessary to restrict them to a specific range. For ease of computation, these values will be processed to conform to a curve that exists between 0 and 1.

To provide the system administrator a degree of control regarding the distribution of values within this curve, he will be able to express an upper significance boundary (USB) and a lower significance boundary (LSB). The values that occur between the LSB and USB will represent 90 percent of the region between 0 and 1 and will be linear. Values that occur outside of that boundary will taper asymptotically toward the overall upper and lower boundaries of 0 and 1.

In order to achieve this effect, two formulas will need to be selectively applied to the unprocessed job parameters. A Central Formula will be applied to values that occur between the LSB and USB range and a Boundary Formula will be applied to values that occur outside of that range.

Note: During the application of these initial formulae, the curve that is created is between -1 and 1. This curve will be adjusted to the proper range (0 to 1) during a later calculation.

The Central Formula

The central formula will force all values between the LSB and USB to be distributed linearly within the central 90% of the curve. Consequently, a parameter that is equal to the lower significance boundary will be assigned a raw weight of -0.9 and a parameter that is equal to the upper significance boundary will be assigned a raw weight of 0.9. The following formula can be used to generate these values:

$$\text{raw weight} = \frac{18x - 9(USB + LSB)}{10(USB - LSB)}$$

The Boundary Formula

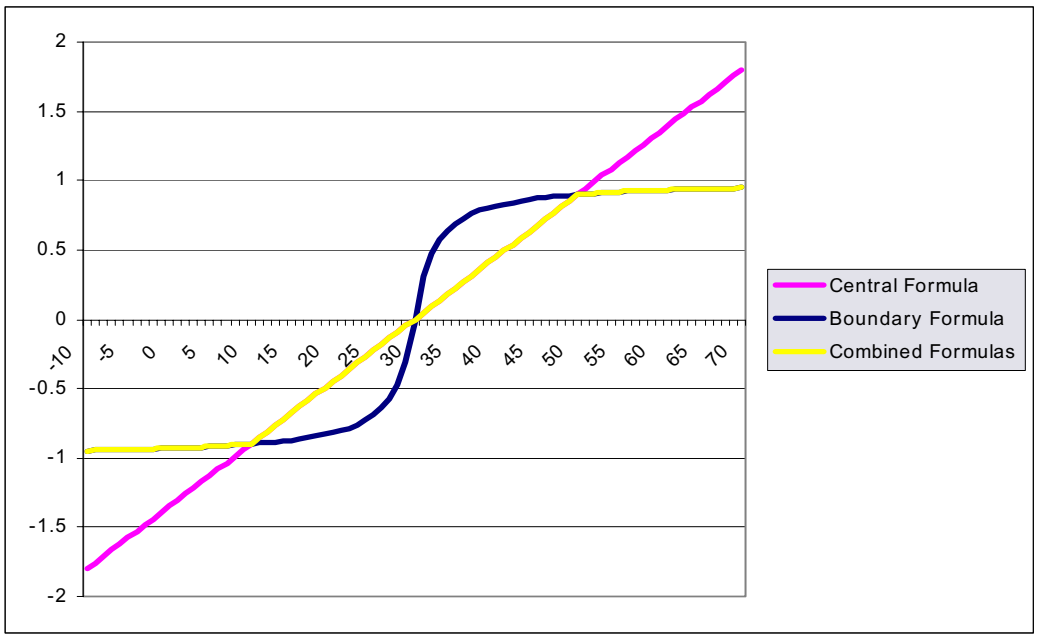
The body formula is applied to all parameter values that fall outside of the range between LSB and USB. In order to form a continuous curve, it is structured to coincide with the weights generated by the central formula at the upper and lower significance boundaries. The following formula is used to generate these values:

$$\text{raw weight} = \frac{2x - (USB + LSB)}{|2x - (USB + LSB)| + \frac{(USB - LSB)}{9}}$$

The Raw Curve

When the results of these two formulae are combined it produces a curve that can be used to represent all of the values for a specific parameter within the range from -1 to 1. The following diagram shows the curves produced by these equations.

Figure 2: Raw Curves Produced by the Central and Boundary Formulae



Computing the Processed Weight Factor

Now that the job's weight has been fitted to a raw curve within a bounded region, it will be necessary to scale this curve to give some factors greater significance than others. Once the raw weights for all sort stages have been computed and scaled they will be multiplied together to produce a total weight for the job.

Because the use of negative values will only impede the computation of weights, the processed weight curves will exist in the range between 0 and 1. To provide greater significance to a specific type of parameter, its overall consumption of this range will be increased or decreased depending on its weight. We'll accomplish this by introducing a *lower weight boundary.*

The Lower Weight Boundary

To change the significance of various sorting stages, the system administrator will apply a positive integer weight to each of the stages. The higher the weight, the greater the impact that stage will have on the scheduling outcome. A weight of 0 indicates that the stage has no effect. This user defined weight factor will be used to determine the range of the processed weight curve for each sorting stage.

The following formula is used to compute the *lower weight boundary* (LWB) for a specific sorting stage. Note that the upper boundary is always 1.

$$\text{lower weight boundary} = \frac{1 - \textit{weight}}{\sum \text{weights}}$$

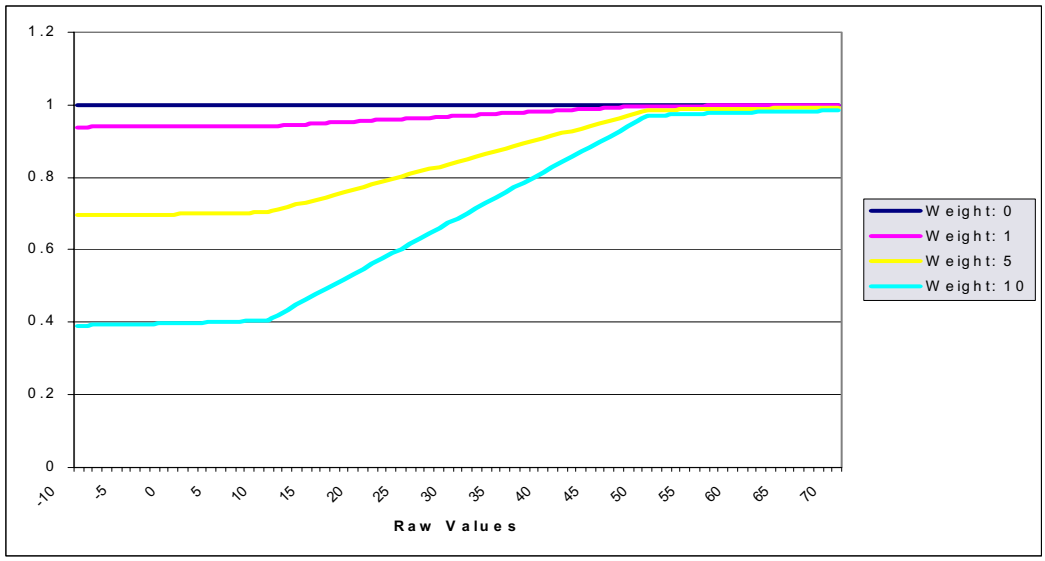
Fitting the Raw Curve to the Weighted Curve

The processed weight curve is generated by compressing the raw curve for the specific sorting stage to fit between the range specified by the stage's lower weight boundary and 1. The following formula can be used to process the raw value.

$$x = \text{raw value}$$
$$LWB = \text{lower weight boundary}$$
$$\text{processed weight value} = ((x+1)/2) * (1 - LWB) + LWB$$

The following chart shows the effect of applying various user-defined weights to identical data. You'll note from the chart that a weight of 0 has resulted in a horizontal line at the value 1, rendering that parameter ineffective.

Figure 3: A Raw Curve After Processing with Various Weights



6. Building a Sorting Stage for the UnderLord

The UnderLordStage Base Class

As mentioned earlier, a sorting stage is merely a C++ class that is derived from the UnderLordStage object. The header file for this base class is provided in the include directory of the source distribution. In order to build a new sorting stage, it will be necessary to derive two methods for your new class: getWeight and compute.

getWeight — *double getWeight (void)*

The getWeight method is called by the scheduler to determine the weight that has been assigned to this particular scheduling stage. The weight can be a static value that you embed in the class or, preferably, it can be a dynamic value that is read from the configuration file when the scheduler is started or reinitialized with a *SIGHUP*.

If you wish to use the configuration file as a repository for storing weights for your sorting stages, you should examine some of the sample stages that are provided with the UnderLord distribution. An API for accessing data in the configuration file is provided in the *PBS_Parameters.h* header file in the include subdirectory of the source distribution. As a simple illustration, the following calls are used to retrieve the weight entry for the User Share Stage.

Figure 4: Reading Weight Factor from Configuration File

```
PBS_Parameters & params =
   PBS_Parameters::attach();

double weight = params.getConfigFloat
   ("User Share Stage", "weight", 1.0);
```

compute — *double compute (PBS_Jobs ** j, size_t cnt, double weightSum)*

The compute method is the workhorse of the sorting stage. It is called with an array of pointers to PBS_Job objects, a count of those objects, and the sum of the weights of all sorting stages that will be applied to these jobs.

The class will compute a *processed weight* for each job in this list. After the weight has been computed, it will call the *getWeight* method of the *PBS_Job* object to obtain the existing weight that was assigned by the other sorting stages. It should multiply the existing weight by the locally generated weight and then call the PBS_Job's *setWeight* method to install the resulting value.

For an extended illustration of how this works, see the source code for one of the sorting stages that were provided with the UnderLord distribution.

Sample Sorting Stage

The following source code illustrates a sorting stage that sorts jobs based on their duration. This stage is provided as part of the UnderLord's source distribution.

Figure 5: DurationStage.h - C++ header file for the Job Duration Stage

```
#ifndef _DURATION_STAGE_H_
#define _DURATION_STAGE_H_ 1

#include <UnderLordStage.h>
#include <PBS_Parameters.h>


class DurationStage : public UnderLordStage
{
public:
   inline DurationStage    (void );
   inline double getWeight (void );
   void          compute   (PBS_Job ** jobs,
                            size_t jobCnt,
                            double weightSum );
};

// ************************************************************
// * DurationStage::DurationStage :
// *This is the constructor for the DurationStage. It
// *calls the constructor for the UnderLordStage object.
// ************************************************************
inline DurationStage::DurationStage ( void )
   : UnderLordStage ()
   {
   }

// ************************************************************
// * DurationStage::getWeight :
// *Returns the weight of this stage in relationship to all
// *other stages.
// ************************************************************
inline double DurationStage::getWeight ( void )
   {
   // ********************************************************
   // * Read the user defined weight for this stage from the
   // * scheduler.cfg file located in /usr/spool/PBS
   // ********************************************************
   double weight =
      PBS_Parameters::attach().getConfigFloat
         ("Job Duration Stage", "weight", 1.0);

   // ********************************************************
   // * Return the weight as a positive floating point value.
   // ********************************************************
   return (weight<0)?0.0:weight;
   }

#endif /* _DURATION_STAGE_H_ */
```

Figure 6: DurationStage.cc - C++ source file for the Job Duration Stage

```
#include <DurationStage.h>
#include <SchedStatic.h>
#include <math.h>

// ************************************************************
// * DurationStage::compute :
// *   This stage of the scheduling algorithm will compute the
// *   weight of an individual job based on the amount of
// *   walltime that it has requested.
// *
// *   jobs      :The array of PBS_Jobs that are being weighted.
// *   jobCnt    :The number of PBS_Jobs in the array.
// *   weightSum :The sum of the weights of all stages that
// *              will be used.
// ************************************************************
void DurationStage::compute
   ( PBS_Job ** jobs, size_t jobCnt, double weightSum )
   {
   // ********************************************************
   // * First requested the upper and lower bounds of
   // * significance from the configuration file.  When each
   // * value is tabulated and fitted to a curve, 90% of the
   // * curve is represented by those values that occur between
   // * the lower significance boundary and the upper
   // * significance boundary.
   // *
   // * Jobs whose durations are shorter than the lower
   // * significance boundary will be fit in the bottom 5% of
   // * the curve, and those whose durations is greater than
   // * the upper significance boundary will be confined to the
   // * top 5% of the curve.
   // *
   // * The time expressed in the configuration file is given
   // * as whole seconds.  By default the lower significance
   // * boundary is 1 minute and the upper significance
   // * boundary is 24 hours.
   // ********************************************************
   PBS_Parameters & params = PBS_Parameters::attach();
   double LSB;
   double USB;

   LSB = SchedStatic::stringToDuration(
      params.getConfigString(
         "Job Duration Stage",
         "lowerBound",
         NULL), 60);
   USB = SchedStatic::stringToDuration(
      params.getConfigString(
         "Job Duration Stage",
         "upperBound",
         NULL), 24*3600);
```

Figure 6: DurationStage.cc (Cont) - C++ source file for the Job Duration Stage

```
// ********************************************************
// * In order to insure a decent curve shape and proper
// * behaviour, The USB and LSB value MUST BE POSITIVE, and
// * the LSB must be less than the  USB. For values that are
// * out of bounds I'm assigning a default of 24 hours for
// * the USB and USB/10 for the LSB.
// ********************************************************
if(USB < 0) USB = 24*3600;
if(LSB<0 || LSB>USB) LSB = USB/10.0;

// ********************************************************
// * Next, request the countProcs flag from the
// * configuration file. It is a yes or no value that
// * indicate whether the number of processors should be
// * considered when calculating the overall job duration.
// ********************************************************
int countProcs = !strcasecmp
   ("yes", params.getConfigString
      ("Job Duration Stage", "countProcs", "yes"));

// ********************************************************
// * Next we must obtain the weight of this stage (relative
// * to all other stages that will be used. By default all
// * stages are given a weight of 1.
// ********************************************************
double weight = getWeight();

// ********************************************************
// * If the stage weight is less than or equal to 0, then
// * the stage will have no effect on the outcome and we
// * will return without computing it.
// ********************************************************
if(weight <= 0 || weightSum <= 0 ) return;

// ********************************************************
// * If the weight is greater than 0, then we will compute
// * the lower weight boundary using this value and the
// * weightSum value.  The LWB indicates the bottom of the
// * curve for this stage... Note: all stages will fit their
// * values to a curve between LWB and 1.0.
// ********************************************************
double LWB = 1.0-(weight/weightSum);

// ********************************************************
// * Now we'll walk through each of the jobs and apply the
// * weight that is appropriate for its job duration.
// ********************************************************
for(size_t idx=0; idx<jobCnt; idx++)
   {
   double rawWeight = 0;

   // ****************************************************
   // * Read the job's duration from the job.
   // ****************************************************
   double duration = jobs[idx]->getDuration();
```

Figure 6: DurationStage.cc (Cont) - C++ source file for the Job Duration Stage

```
      // ******************************************************
      // * If countProcs is TRUE, then multiply the duration
      // * times the number of processors requested.
      // ******************************************************
      if(countProcs) duration*=jobs[idx]->getProcCnt();

      // ******************************************************
      // * If the duration is between the lower significance
      // * boundary and the upper significance boundary, then
      // * we'll use the 'Central Formula' to compute its
      // * weight.
      // ******************************************************
      if(LSB<=duration && duration<=USB)
         {
         rawWeight =-(18.0*duration - 9.0*(USB+LSB)) /
                       (10.0*(USB-LSB));
         }
      // ******************************************************
      // * Otherwise use the boundary formula. Note the
      // * variable t is used to reduce the compute time for
      // * the operation and to simplify the appearance of the
      // * code.
      // ******************************************************
      else
         {
         double t = (2.0*duration - (USB+LSB));
         rawWeight = -(t / (fabs(t)+(USB-LSB)/9));
         }

      // ******************************************************
      // * Next use ths raw weight in conjunction with the
      // * lower weight boundary to fit the curve within the
      // * desired region and obtain the processed weight of
      // * this job in this stage.
      // ******************************************************
      double procWeight = ((rawWeight+1.0)/2.0)*(1-LWB)+LWB;

      // ******************************************************
      // * Multiply this value against the current weight of
      // * the job to introduce this stage's influence on the
      // * overall weight of the job. Note that if the initial
      // * weight is less than (or equal to) 0, it is restored
      // * to 1 - making this stage the only one of any
      // * significance.
      // ******************************************************
      double jobWeight;

      if((jobWeight = jobs[idx]->getWeight())<=0)
         {
         jobWeight=1.0;
         }
      jobs[idx]->setWeight(jobWeight * procWeight);
      }
   }
```

Adding the New Stage to the UnderLord

After the code for the new sorting stage is completed, the next step is to add it to the list of sorting stages that will be used by the UnderLord. This is accomplished by modifying the source code *SchedMain.cc* in the UnderLord source distribution.

In the *schedule* function of this file, the UnderLord object is initially created and then assigned to the underlord pointer. After construction, the developer should call the *addStage* method to add a new sorting stage to the UnderLord. The example below shows an excerpt from the schedule method with the critical code for adding the new JobDurationStage to the scheduler.

Note: Once a scheduling stage has been allocated and assigned to the scheduler, it becomes the property of the UnderLord and should not be directly accessed afterwards. These objects will be automatically destroyed when the scheduler is shutdown or reinitialized.

Figure 7: Excerpts from SchedMain.cc - Adding a new stage to the UnderLord

```
extern "C" int schedule(int cmd, int sd)
   {
   ...
   if(underlord == NULL)
      {
      underlord = new UnderLord();
      underlord->addStage(new DurationStage());
      }
   ...
   }
```

7. Match-Making: Periods of Availability on a Single Node

Match-Making Objectives

One of the most difficult tasks in designing a scheduling system is that of matching the waiting jobs to the available resources. The scheduler must balance the objective of finding the 'perfect match' for optimal system performance against job priorities established during the sorting stage and still be able to complete this analysis in a reasonable amount of time. In the end, the goal of match-making is to maintain high system utilization, conform to the site's scheduling policies, and not starve jobs with unusual or large system requirements --- *t'aint a simple problem*.

The MatchMatrix Solution

The UnderLord uses an extensive set of C++ classes to develop an object-oriented solution to matchmaking. Unfortunately, the complexity of this system of objects prevents them being fully explained here. For those interested, an OMT object model illustrating the structure and organization of these classes is available from the following address:

http://www.jlab.org/hpc/Underlord/MatchMatrix.pdf

Populating the ClusterNode Object with Running Jobs

At the beginning of the scheduling cycle, the UnderLord will start by collecting a list of available nodes (those that are not *down, offline*, *reserved* or in an *unknown* state. It will read the attributes and the number of processors of these nodes (as defined in the PBS configuration files) and will use this information to construct a C++ ClusterNode object. The ClusterNode object is the key interface that the scheduler will use to perform match-making.

The next step in this process is to determine the *periods of availability* on each node. This is done by obtaining a list of executing jobs and assigning these jobs to the ClusterNode objects. Since there is no way to be certain when a job will really end, we have to rely on the information provided by the user to project the job's duration. Once all running jobs have been assigned to the ClusterNodes, there will be empty spaces where jobs are not actively being executed... these are called *periods of availability.*

Periods of Availability

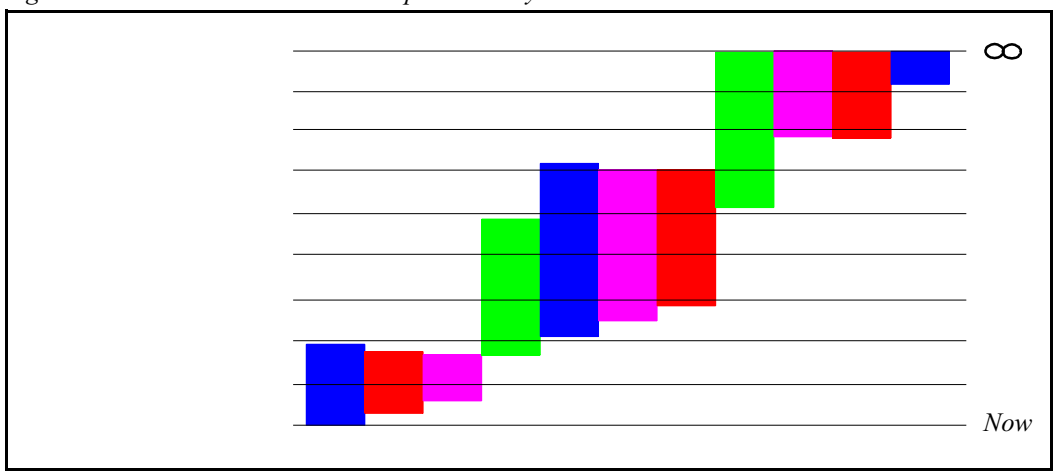
Due to the dynamic nature of batch processing, the duration and start times of these *periods of availability* can be any shape or size. The only certainty is that the last period of availability on any node will start after the last job completes and will extend to infinity. While obvious enough, this is useful information because it means that, at some point in the future, the entire cluster will be available and is schedulable. Consequently, any job that the cluster is capable of running should be schedulable regardless of the current number or type of jobs in execution.

In the event that a job requires more than one processor per node, then the UnderLord must provide *overlapping periods of availability* that represent free time across multiple processors. The following series of illustrations will demonstrate the algorithm that the UnderLord uses to catalog *overlapping periods of availability.*

Finding Overlapping Periods of Availability

Assume that the following collection represents the *free time slots* available on the 4 processors of a single node - each color representing one of the processors.

Figure 8: Available time slots on a 4 processor system



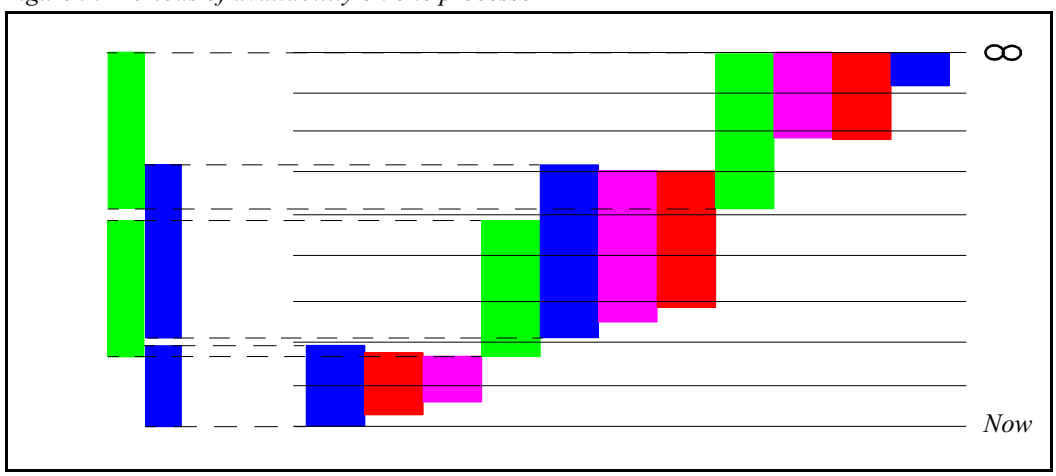
After extracting these *periods of availability*, the UnderLord has placed them into an ordered list (ordered by start time, then duration). Once this list has been created, the scheduler may then walk through it and create a list of *periods of availability* that extend across one or more processors.

Note that the UnderLord will not maintain a list of redundant periods. In the event that one period is completely overlapped by another, the second period will be disregarded. This has no ill effect on scheduling since a job will only occupy a node once, regardless of the number of processors available on that node.

Periods of Availability on One Processor

The diagram below illustrates the four *periods of availability* that can be extracted from this node when only one processor has been requested.

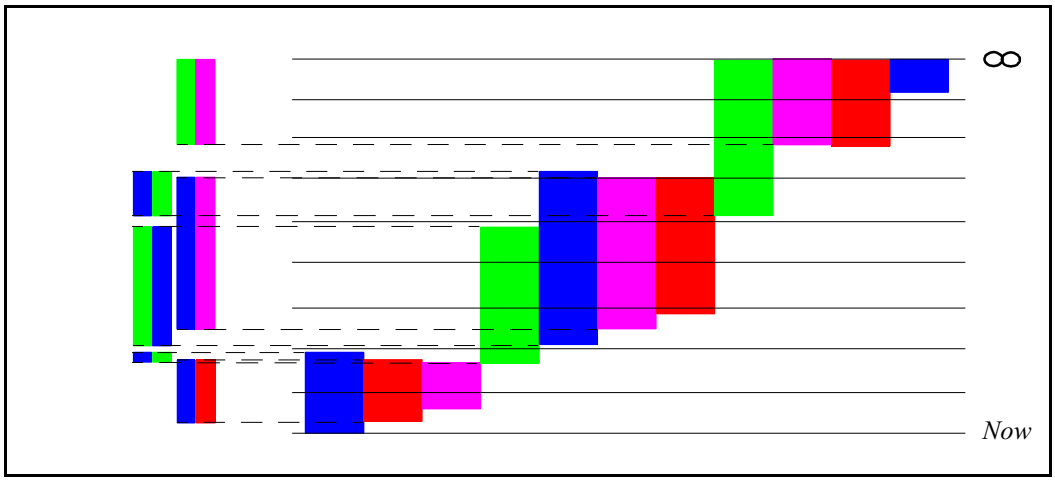
Figure 9: Periods of availability on one processor



Periods of Availability on Two Processors

When the number of processors requested increases to two, the number of *periods of availability* increases to six, but the duration of each period decreases somewhat. Additionally, the beginning of the earliest, infinite period of availability becomes later.

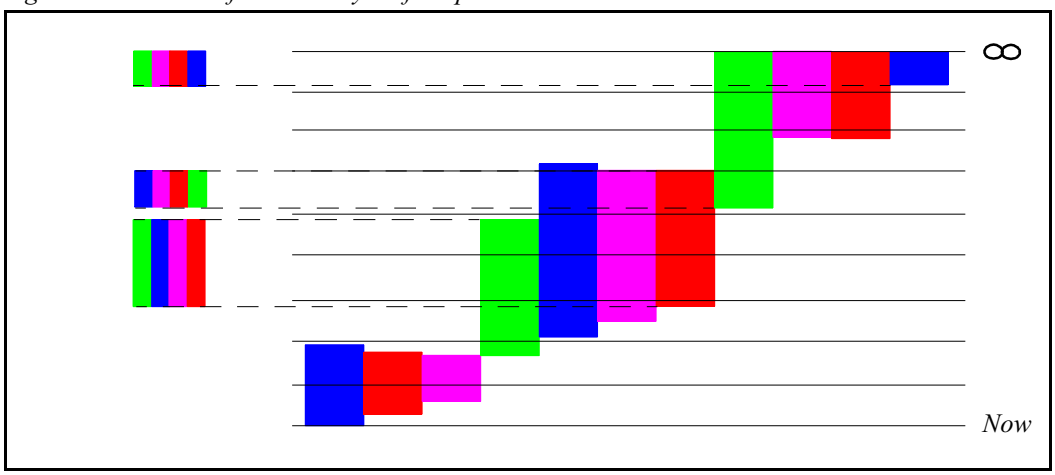
Figure 10: Periods of availability on two processors



Periods of Availability on Four Processors

This effect becomes more apparent when the UnderLord extracts *periods of availability* in groups of four processors.

Figure 11: Periods of availability on four processors



Improving the Performance of these Operations

As you might imagine, the initial generation of these lists can be a very time consuming process. This situation is worsened if you are working with a system that has many processors per node and, consequently, a greater number of permutations.

The UnderLord optimizes these operations by only generating each group of lists when a job requests a specific number of processors. Additionally, when a new job is scheduled on a node, each of its lists of *periods of availability* is marked as *dirty*. The list is only refreshed when another job with the same number of requested processors needs to be processed.

For example, imagine that your queue contains one job that required four processors per node and ten jobs that require two processors per node. The list of *periods of availability* for four processors will be created once, used and then never updated. The list of *periods of availability* for two processors will be created and then will be incrementally updated each time a job is assigned to one of the nodes. The lists for one and three processors will never be created.

As an additional optimization, at the beginning of the scheduling cycle the UnderLord will determine the duration of the briefest job in the queue. When there is no immediate period of availability (all nodes are active) or when the duration of the longest immediate period of availability is shorter than the shortest job, then the scheduling cycle will be terminated and no other jobs will be evaluated for placement.

Note: In the event that the system administrator is using either the execSummary or the stageSummary option, whenever the summary period expires the scheduler will compute the projected execution time for every job in the system. For this reason it is important to make the summaryRate high enough that system performance will not be impacted by these operations. The default summaryRate that is specified in the UnderLord's configuration file is ten minutes.

8. Match-Making: Periods of Deployment on Multiple Nodes

Parallel Objectives

When scheduling jobs in a simple batch environment, it is not necessary to address the issue of scheduling concurrent time across multiple nodes. However, in parallel environments this is a critical element of scheduling. In the UnderLord scheduler we have chosen to use a histogram approach to scheduling on multiple nodes that is very similar to that used to find overlapping *periods of availability* on multi-processor nodes.

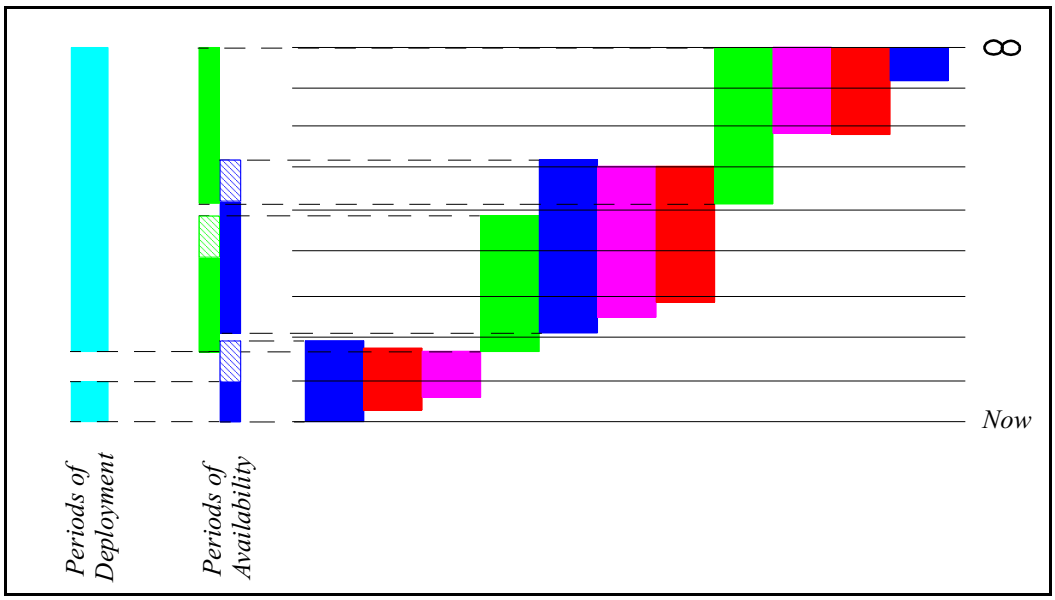
When scheduling concurrent time on multiple nodes, the *periods of availability* are still generated for each node in the system. Once these lists are generated, the scheduler will begin querying each node for a list of time slots where the job can fit. ***This approach is different than you might expect...***

Earlier, we generated simple lists of periods when the node was available and returned them as time intervals. When performing this part of the match-making process, we will pass the *projected duration* of each job to the scheduler and it will return ***a list of time intervals in which the job may be started and still complete within the period of availability on that node***. These intervals are called the *periods of deployment*.

Periods of Deployment

The illustration below shows the *periods of availability* on a node, and the *periods of deployment* that are returned for a job that has a projected duration of one hour. Note that each line on the chart represents one hour of walltime.

Figure 12: Periods of deployment on one processor

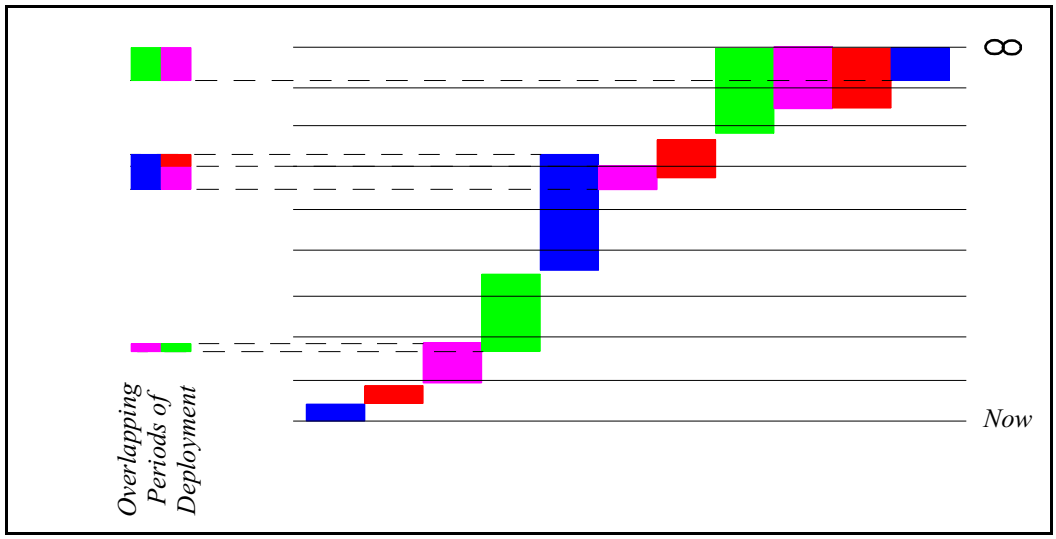


In the illustration above, you'll note that while there are four *periods of availability* there are only two *periods of deployment*. This is because the period of deployment is not associated with a specific processor on the node --- that information will be garnered when the job is sent to the node to be scheduled. If two *periods of availability* overlap sufficiently (by at least the duration of the job) then the period of deployment is continuous. Using this approach may greatly decrease the number of time intervals that must be evaluated during the scheduling cycle.

Once the *periods of deployment* have been obtained from all available nodes, these periods will again be linked together into an ordered list (ordered by start time and sub-ordered by duration). The scheduler can then walk this list to locate the earliest group of nodes that are sufficient to run the job.

Note that since each of these time intervals represents a period in which the job may be *started*, only the most minimal overlap of these periods is required. The diagram below shows how the *periods of deployment* on four nodes are evaluated to find the earliest time that a job can be executed.

Figure 13: Overlapping periods of deployment on four nodes



The earliest time interval in which the job can be started is the small magenta/green combination at the bottom. While this may not appear to be very much time, remember that these regions represented the available *start times* only.

9. Match-Making: Jobs Requiring Multiple Node Types

What are Node Types?

Each node in a batch cluster has specific attributes associated with it. A node with a particularly large data disk may have a BIGDISK attribute, while one with 512 MBytes of RAM may have a MEM512 attribute. The Portable Batch System allows you to specify the number and types of nodes that a job requires.

The UnderLord allows the caller to request up to 32 different node types for each job. The scheduler will make an array of *periods of deployment* for each node type. It will then collect the number and types of nodes from these lists, using the same histogram approach that was described earlier.

Pitfalls of Multiple Node Types

I should note that there are scenarios when requesting multiple node types when the scheduler might fail to satisfy the request, even though there are sufficient nodes available. For instance, if several of the nodes have overlapping node types, the scheduler might assign a BIGDISK/MEM512 node to the BIGDISK list of nodes and, inadvertently, not have sufficient MEM512 nodes to do the job. In an attempt to address this, if the scheduler cannot locate sufficient nodes for a job with multiple node types, it will reorder the node types and try again... hoping for a better outcome. The user can improve the chances of successfully deploying multiple node type jobs by listing the rarest node types first.

Because of the inherent difficulty and time-consuming nature of deploying jobs with multiple node types, many scheduling systems do not support it. We intend to develop a more comprehensive solution to this problem in future versions of this product.

10. Future Directions

Objectives

As mentioned earlier, the UnderLord was originally designed as the site level scheduler for a multi-site meta-facility. The design, construction and implementation of this meta-facility remains one of our primary objectives. Many of our other goals for future development are in line with achieving this overall mission.

The following is a list of issues that we intend to address in future releases.

1. Enhance reliability of the scheduler through continued testing on increasingly larger systems with a more diverse job mix.
2. Introduce an easy interface for developers to reserve periods of time on a node or nodes.
3. Allow the system administrator to reserve periods of time during which only certain jobs or certain classes of jobs can be run.
4. Develop a 'bag of nodes' solution. In this design, whenever a user requests a specific number of nodes (8, 16, 32, 64), they will always be allocated in predefined groups. This solution is being developed to ensure the optimal performance of high speed interconnects that are being used in our cluster.
5. Develop a rating system that allows the scheduler to identify the cost of moving a job to another site for processing. This would include the cost of relocating data files as well as simply relocating the job.
6. Create XML based reporting tools that generate summaries of how and why the UnderLord is scheduling jobs. ASCII text summary and projection files are already being generated, however, XML files can easily be manipulated and displayed on web pages.
7. Develop a graphical user interface allowing the system administrator to easily modify and update scheduler parameters to dynamically tune the behavior of the system.
8. Allow the user to specify optional node combinations and durations, i.e: 32 nodes for 1 hour, or 16 nodes for 2 hours, etc.
9. Develop an improved algorithm for handing requests that include multiple node types.