Workload Management with LoadLeveler

- Detailed description of new features
- Focus on internal and external schedulers
- Example scenarios

Subramanian Kannan
Mark Roberts
Peter Mayes
Dave Brelsford
Joseph F Skovira

ibm.com/redbooks
Take Note! Before using this information and the product it supports, be sure to read the general information in “Special notices” on page 199.

First Edition (November 2001)

This edition applies to Parallel System Support Program Verison 3, Release 4 and LoadLeveler Version 3 Release 1 for use with the AIX Operating System Version 5 Release 1

Note: This book is based on a pre-GA version of a product and may not apply when the product becomes generally available. We recommend that you consult the product documentation or follow-on versions of this redbook for more current information.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. JN9B Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

© Copyright International Business Machines Corporation 2001. All rights reserved.
Note to U.S Government Users – Documentation related to restricted rights – Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.
Contents

Figures ................................................................. ix
Tables ................................................................. xi
Preface ................................................................. xiii
The team that wrote this redbook ................................... xiii
Special notice ......................................................... xv
IBM trademarks ...................................................... xv
Comments welcome ................................................ xv

Chapter 1. LoadLeveler basics ...................................... 1
1.1 Overview ......................................................... 2
1.2 LoadLeveler filesets ............................................. 2
1.3 Configuration files .............................................. 2
1.4 Daemons ......................................................... 4
1.5 Basic commands ............................................... 5
1.6 Graphical user interface (GUI) .............................. 5
1.7 Application programming interfaces (APIs) ............... 6
1.8 Simple examples of job command files ..................... 7

Chapter 2. Easy installation ......................................... 9
2.1 Quick steps to install .......................................... 10
2.1.1 Install the product on all the nodes .................... 10
2.1.2 Set up the loadl user ..................................... 11
2.1.3 Create /var/loadl file system ........................... 11
2.1.4 Change the ownership .................................... 12
2.1.5 Run the LoadLeveler installation script ............... 12
2.1.6 Create links for config file .............................. 13
2.1.7 Execute llextSDR ......................................... 14
2.1.8 Edit LoadL_admin ......................................... 14
2.1.9 Edit LoadL_config ....................................... 15
2.1.10 Start LoadLeveler ....................................... 15

Chapter 3. New features ............................................ 17
3.1 Task assignment ............................................... 18
3.2 Consumable resources and WLM integration ............. 18
3.3 gsmonitor daemon ............................................ 20
3.4 Process tracking ............................................... 20
3.5 Distributed computing environment (DCE) ................ 21
3.6 64-bit support .................................................. 21
3.7 GANG scheduling ............................................. 21
3.8 Preemption ..................................................... 23
3.9 Checkpoint/restart ............................................ 24
3.10 Adapter striping .............................................. 24

Chapter 4. Managing jobs ......................................... 27
4.1 Job steps ....................................................... 28
4.2 Job filters ...................................................... 35
4.3 Run-time environment variables ............................... 36
4.4 Managing the job environment ................................. 37
4.5 Using job command file variables .............................. 38
4.6 Managing classes ............................................. 39
4.7 Managing limits ............................................... 45
4.8 Managing job queues ......................................... 47
4.9 Managing job execution ...................................... 50
4.10 Using prolog and epilog scripts .............................. 53

Chapter 5. Managing parallel jobs ................................. 57
5.1 Task assignment ............................................... 58
  5.1.1 Node and total_tasks .................................... 58
  5.1.2 Node and tasks_per_node ................................. 58
  5.1.3 Task geometry ........................................... 59
  5.1.4 Blocking .................................................. 60
5.2 Interactive parallel ........................................... 61
  5.2.1 POE environment variables ............................... 61
  5.2.2 Use of pools .............................................. 61
  5.2.3 Debugging ............................................... 63
  5.2.4 Use of IP ................................................ 65
  5.2.5 Using LoadLeveler command files ...................... 66
5.3 Batch parallel ............................................... 68
  5.3.1 MPI - User space in shared mode ....................... 69
  5.3.2 MPI - User space in non-shared mode .................. 69
  5.3.3 MPI - IP mode over the switch ......................... 70
  5.3.4 MPI - IP mode over ethernet ........................... 71
5.4 LAPI .......................................................... 72
5.5 Managing SMP nodes ......................................... 73
  5.5.1 Consumable resources .................................. 73
  5.5.2 Node usage: to share or not to share .................. 79
  5.5.3 Using shared memory ................................... 80

Chapter 6. Checkpoint and restart ............................... 81
6.1 Overview ..................................................... 82
6.2 Job types .................................................... 82
6.3 Checkpointing types ............................................. 83
  6.3.1 System initiated checkpoints .............................. 83
  6.3.2 Complete user initiated checkpoints .................... 84
  6.3.3 Partial user initiated checkpoints ........................ 84
6.4 Configuration .................................................. 84
  6.4.1 Administration file settings ............................. 84
  6.4.2 Configuration file settings ............................... 84
  6.4.3 Job command file settings ............................... 85
  6.4.4 Environment variables ................................... 85
  6.4.5 Job command file keywords .............................. 85
  6.4.6 Environment variable effects ............................ 88
6.5 Serial ......................................................... 88
  6.5.1 Scenario 1 - Checkpoint and restart on the same node 88
  6.5.2 Scenario 2 - Checkpoint and restart .................... 90
  6.5.3 Scenario 3 - Restarting from a previous checkpoint .... 91
6.6 Batch parallel .................................................. 92
  6.6.1 Scenario 1 - Checkpoint ................................ 92
  6.6.2 Scenario 2 - Restarting from a checkpoint file ........ 94
6.7 Considerations for checkpointing .............................. 94

Chapter 7. Internal schedulers ................................. 97
  7.1 Queuing priorities ........................................... 98
  7.2 Node selection ............................................... 102
  7.3 Requirements and preferences .............................. 103
  7.4 Default scheduler ........................................... 104
    7.4.1 How it works ............................................ 104
  7.5 Backfill .................................................... 105
  7.6 GANG ........................................................ 106
    7.6.1 Introduction to GANG scheduling ....................... 106
    7.6.2 The GANG Matrix ........................................ 106
    7.6.3 New GANG scheduler commands ......................... 108
    7.6.4 Preemption using the GANG scheduler .................. 109
    7.6.5 Class starting rules using the GANG scheduler ....... 110
    7.6.6 GANG scheduling keywords ............................. 110
    7.6.7 Examples of the GANG scheduler ....................... 112
    7.6.8 Comparison of the internal schedulers ............... 114

Chapter 8. Customized schedulers using API ................. 115
  8.1 LoadLeveler API .............................................. 116
    8.1.1 Data access API ......................................... 116
  8.2 External schedulers .......................................... 120
    8.2.1 Maui ..................................................... 120
    8.2.2 Special Resource Scheduler (SRS) ....................... 132
Chapter 9. Resource usage and monitoring ................................. 139
  9.1 Accounting ..................................................................... 140
    9.1.1 Job account validation ............................................. 142
    9.1.2 API for custom reports ............................................. 144
    9.1.3 Using CPU speed .................................................... 145
    9.1.4 Interaction with LL/SP/AIX accounting ...................... 146
  9.2 Monitoring ..................................................................... 147
    9.2.1 Customizing the LoadLeveler GUI ............................. 147
    9.2.2 Customizing llq displays ......................................... 151
    9.2.3 Other monitoring tools ............................................ 152
  9.3 Mail notification .......................................................... 155
  9.4 Managing log files ........................................................ 161
    9.4.1 What are the log files and locations ......................... 161
    9.4.2 How to save log files .............................................. 161
    9.4.3 Adding debug output to the log files ....................... 162
    9.4.4 How to track a job through the system ..................... 164
    9.4.5 Other relevant log files .......................................... 165
  9.5 Troubleshooting .......................................................... 167
    9.5.1 Managing LoadLeveler files ................................... 167
    9.5.2 Schedd node failure .............................................. 168
    9.5.3 Job not running .................................................... 168
    9.5.4 Job command files ................................................ 169
Appendix A. Tools ................................................................ 171
  llbeans.c .......................................................................... 172
  ll_get_nodes.c .................................................................. 179
  Example program hist.c ..................................................... 182
  Example program llacct.c ................................................... 183
  Header and Makefiles for getting history data ..................... 185
Appendix B. Concise reference ............................................... 187
  LoadLeveler environment variables .................................... 188
  LoadLeveler variables ...................................................... 188
  Job command file variables ............................................. 189
  Job command file keywords ............................................. 189
  Job states ........................................................................ 190
  Daemon States ............................................................... 191
    Schedd ......................................................................... 191
    Startd. ......................................................................... 191
  Admin file ....................................................................... 191
  Debug flags ..................................................................... 193
  llctl subcommands ......................................................... 194
  llstatus -f/-r field selectors ............................................ 194
Figures

5-1 Managing CPUs using LoadLeveler and WLM .......................... 77
5-2 Managing memory using LoadLeveler and WLM ....................... 78
7-1 Sample GANG Matrix ................................................. 107
8-1 Job queue ................................................................. 135
8-2 Node resources ....................................................... 136
8-3 SRS resource allocation for jobs-scenario 1 ............................ 137
8-4 SRS resource allocation for jobs-scenario 2 ............................ 138
8-5 Final job allocation map ............................................. 138
9-1 xloadl showing customized output .................................. 150
9-2 NERD application .................................................... 153
9-3 LoadView application .................................................. 154
9-4 Llama application ..................................................... 155
# Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>LoadLeveler filesets</td>
<td>2</td>
</tr>
<tr>
<td>1-2</td>
<td>Summary of LoadLeveler commands</td>
<td>5</td>
</tr>
<tr>
<td>2-1</td>
<td>LoadLeveler initial setup</td>
<td>12</td>
</tr>
<tr>
<td>3-1</td>
<td>Simple example of a GANG Matrix</td>
<td>22</td>
</tr>
<tr>
<td>4-1</td>
<td>How limits are set</td>
<td>46</td>
</tr>
<tr>
<td>5-1</td>
<td>Task assignment - node and total_tasks</td>
<td>58</td>
</tr>
<tr>
<td>5-2</td>
<td>Task assignment - blocking</td>
<td>60</td>
</tr>
<tr>
<td>5-3</td>
<td>Task assignment - unlimited blocking</td>
<td>60</td>
</tr>
<tr>
<td>5-4</td>
<td>MP_INFOLEVEL options</td>
<td>63</td>
</tr>
<tr>
<td>5-5</td>
<td>WLM CPU allocation</td>
<td>76</td>
</tr>
<tr>
<td>6-1</td>
<td>Checkpoint command line options</td>
<td>83</td>
</tr>
<tr>
<td>6-2</td>
<td>Effects of checkpoint values</td>
<td>88</td>
</tr>
<tr>
<td>7-1</td>
<td>LoadLeveler priorities</td>
<td>100</td>
</tr>
<tr>
<td>7-2</td>
<td>Job priority values extracted from llq command</td>
<td>101</td>
</tr>
<tr>
<td>8-1</td>
<td>LoadLeveler APIs</td>
<td>116</td>
</tr>
<tr>
<td>8-2</td>
<td>Data access API subroutines</td>
<td>116</td>
</tr>
<tr>
<td>9-1</td>
<td>Affects of cpu_speed on accounting job times</td>
<td>146</td>
</tr>
<tr>
<td>9-2</td>
<td>The effect of adding debug flags</td>
<td>162</td>
</tr>
</tbody>
</table>
Preface

Scientific and technical computing applications, in general, consist of Parallel and Serial programs. In an RS/6000 SP environment, the IBM LoadLeveler product provides various means to schedule and manage these applications on a set of nodes. In this book, we provide a practical approach for scheduling and managing the LoadLeveler environment. We describe quick installation and configuration procedures for setting up your LoadLeveler environment and then describe, in detail, how to exploit the various LoadLeveler features for job and resource management. We also provide you with example scenarios, sample configurations, and guidelines on how to use several new features supported in the LoadLeveler product.

For the first time in any book, we will show you how to use external schedulers like MAUI and SRS, which are developed using the LoadLeveler application programming interface (API). We also focus on smart ways to use the LoadLeveler keywords and environment variables to customize some of the administration tasks to suit your environment.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Poughkeepsie Center.

Subramanian Kannan is a Project Leader at the International Technical Support Organization, Poughkeepsie Center. He leads RS/6000 SP and Cluster system related projects at ITSO and contributes to writing redbooks and teaches IBM classes worldwide on all areas of RS/6000 SP and Clustered servers. Before joining the ITSO two years ago, Kannan worked in ESG team, India as Advisory IT specialist.

Peter Mayes works for IBM as a Senior IT Specialist in the UK. He has 15 years of experience in the field of high-performance computing. He holds the degrees of MA in Mathematics, MSc in Mathematical Modeling and Numerical Analysis, and DPhil in Engineering Mathematics, all from the University of Oxford. His areas of expertise include FORTRAN programming, particularly for high-performance and parallel computers, and administration of RS/6000 SPs.
Mark Roberts is a Systems Administrator in the UK. He has five years of experience in the High Performance Computing field and has worked at AWE Aldermaston for 13 years. His areas of expertise include AIX and RS/6000 SP system management.

Dave Brelsford is an Advisory Software Engineer in the USA, with more than 25 years experience working for IBM, predominantly in software development. He has a Bachelors degree from Rensselaer Polytechnic Institute in computer science and is currently a member of the LoadLeveler development team.

Joseph F Skovira joined IBM in 1979 and has been working with parallel processors since 1987. As part of the team responsible for the deployment and operation of the SP2 at the Cornell Theory Center, he helped introduce the first BACKFILL Scheduler to the IBM SP2 in 1995. This work led to the development of the LoadLeveler external scheduling API. Involved with LoadLeveler and parallel job schedulers for the past several years, he is also interested in parallel algorithms for genomic applications, signal processing, and computational neuroscience.

Special thanks to Andy Pierce, IBM center, Cornell Theory Center, for his suggestions and guidance during this project.

Thanks to the following people for their contributions to this project:

International Technical Support Organization, Poughkeepsie
Al Schwab, Ella Buslovich, Mike Schwartz, Pete Bertolozzi

International Technical Support Organization, Austin
Matthew Parente and Wade Wallace

LoadLeveler/SP Development, Poughkeepsie Center
April Brown, Paula Trimble, Iris Harvey, Thong Vukhac, Dennis Weinel, Brian Croswell, Rich Coppinger, Waiman Chan, Jerald E Gerner, Joseph McKnight, Gary Mincher, Carol Sutcliffe, Gareth S Bestor

IBM UK
Will Weir

AWE plc, Aldermaston, UK
Phil Renwick, Mary Cushion, David Youns

We would like to thank the following customers and IBM internal LoadLeveler administrators for taking the time to participate in our survey:

Nick Cardo, National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, USA
Barry Schaudt, University of Minnesota Supercomputing Institute, USA
Eric Sills, North Carolina Supercomputing Center, USA
Caroline Pasti, National Centers for Environmental Prediction, USA
Moe Jettie, Lawrence Livermore National Laboratory, USA
Scott Studham, Dave Jackson, Scott Jackson, Pacific Northwestern National Laboratory, USA
Steve Linehan, IBM Poughkeepsie, USA
Marty Gustavson, IBM Rochester MN, USA

Special notice

This publication is intended to help system managers and administrators who are involved in managing LoadLeveler Clusters in an RS/6000 SP or RS/6000 and IBM pSeries server environment. The information in this publication is not intended as the specification of any programming interfaces that are provided by IBM LoadLeveler. See the PUBLICATIONS section of the IBM Programming Announcement for IBM LoadLeveler for more information about what publications are considered to be product documentation.

IBM trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

- AFS®
- AIX®
- AIX 5L™
- BookMaster®
- e (logo)®
- IBM ®
- LoadLeveler®
- Perform™
- pSeries™
- Redbooks™
- Redbooks Logo
- RS/6000®
- SP™
- SP2®
- WebSphere®

Comments welcome

Your comments are important to us!
We want our IBM Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online Contact us review redbook form found at:
  
  \[ \text{ibm.com/redbooks} \]

- Send your comments in an Internet note to:
  
  \[ \text{redbook@us.ibm.com} \]

- Mail your comments to the address on page ii.
LoadLeveler basics

In this chapter, we give a brief overview of the configuration of a LoadLeveler cluster. The following topics are covered:

- Basic LoadLeveler concepts
- LoadLeveler product filesets
- LoadLeveler configuration files
- LoadLeveler daemons
- Basic LoadLeveler administration commands
- Graphical user interface for administration
- LoadLeveler Application programming interfaces (APIs)
- Simple job command files
1.1 Overview

LoadLeveler manages both serial and parallel jobs over a cluster of servers. This distributed environment consists of a pool of machines or servers, often referred to as a LoadLeveler cluster. Machines in the pool may be of several types: Desktop workstations available for batch jobs, usually when not in use by their owner, dedicated servers, and parallel machines.

Jobs are allocated to machines in the cluster by a scheduler. The allocation of the jobs depends on the availability of resources within the cluster and various rules, which can be defined by the LoadLeveler administrator. A user submits a job using a job command file, and the LoadLeveler scheduler attempts to find resources within the cluster to satisfy the requirements of the job. At the same time, it is the job of LoadLeveler to maximize the efficiency of the cluster. It attempts to do this by maximizing the utilization of resources, while at the same time minimizing the job turnaround time experienced by users.

1.2 LoadLeveler filesets

Table 1-1 shows the LoadLeveler filesets required for the various machines within the cluster.

<table>
<thead>
<tr>
<th>Fileset</th>
<th>Description</th>
<th>Installed on</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoadL.full</td>
<td>LoadLeveler</td>
<td>All machines running LoadLeveler daemons</td>
</tr>
<tr>
<td>LoadL.so</td>
<td>Submit-only</td>
<td>Submit-only machines</td>
</tr>
<tr>
<td>LoadL.msg.lang</td>
<td>Language specific messages/man pages</td>
<td>Required by LoadL.full and LoadL.so</td>
</tr>
<tr>
<td>LoadL.html</td>
<td>HTML documentation</td>
<td>Optional</td>
</tr>
<tr>
<td>LoadL.pdf</td>
<td>PDF documentation</td>
<td>Optional</td>
</tr>
<tr>
<td>LoadL.tguides</td>
<td>Task Guides</td>
<td>Optional</td>
</tr>
</tbody>
</table>

1.3 Configuration files

The definition of the resources available within the LoadLeveler cluster and the run-time behavior of the cluster are determined by the LoadLeveler configuration files. By default, these are located in the home directory of the LoadLeveler user ID. The default LoadLeveler user ID is loadl and the home directory is /u/loadl.
Chapter 1. LoadLeveler basics

- **LoadL_admin**
  This is the administration file that defines the machines and their network adapters together with job classes, users, and groups.

- **LoadL_config**
  This global configuration file contains default settings for all the machines in the cluster, such as:
  - Path names to specify log, spool, and history file locations.
  - Daemon port numbers.
  - Configuration parameters for the individual daemons.
  - Administrative users.
  - Job and user epilog/prolog locations.
  - Scheduler type and associated operational parameters.
  - Parameters for ordering the job queues, and the machines within the cluster.

- **LoadL_config.local**
  This local configuration file can be used on individual machines to override the global settings contained in LoadL_config.

The default configuration file location can be overridden using the master configuration file /etc/LoadL.cfg. As well as specifying an alternate location for the configuration files, this file can be used to issue commands on other clusters by setting the environment variable LOADL_CONFIG to the name of a master configuration file defining the other cluster.

*Example 1-1 /etc/LoadL.cfg specifying an alternative cluster configuration*

```
LoadL_Userid = loadl
LoadL_Group = loadl
LoadLConfig = /u/loadl/CONFIG/MARKR/TEST1/LoadL_config
```

We have found it very useful to be able to switch quickly between configurations by changing the single line LoadLConfig (shown in Example 1-1) to point to a completely separate configuration. This could be used in a production environment for testing new configurations with minimum disruption.
1.4 Daemons

The management of jobs and resources within the cluster is accomplished by the various LoadLeveler daemons. The LoadLeveler administrator can configure which daemons should run on which machines, since not all machines within the cluster need to run all the daemons. Only the daemons specified in the configuration files are started on the machines during LoadLeveler startup. The relevant daemons are:

- **LoadL_master**
  The master daemon runs on all machines in the cluster, except submit-only machines. The master daemon manages the other LoadLeveler daemons on the local machine.

- **LoadL_schedd**
  The schedd daemon receives job submissions from `llsubmit` and schedules the jobs to machines allocated by the negotiator daemon.

- **LoadL_startd**
  The startd daemon is responsible for informing the negotiator daemon of local job and machine resource usage, acting upon job requests from remote machines. It is startd that spawns a starter process when instructed by the scheduler daemon to begin a job.

- **LoadL_starter**
  The starter daemon runs the jobs and returns status information regarding the job's processes back to the startd daemon.

- **LoadL_negotiator**
  The negotiator daemon runs on the Central Manager. It receives status information from every schedd and startd in the cluster, maintaining both job and machine information in memory. Only the Central Manager responds to requests for status information from commands such as `llq` and `llstatus`.

- **LoadL_GSmonitor**
  The gsmonitor daemon may run on any machine in the cluster. Its function is to quickly notify the negotiator when a machine becomes unavailable, using the Group Services API available in Parallel System Support Programs (PSSP).

- **LoadL_kbdd**
  The kbdd daemon monitors mouse and keyboard activity on the machine. Once activity is detected, the startd daemon is notified.

A more detailed description of the daemons is given in *Loadleveler V2R2 for AIX: Using and Administering*, SA22-7311.
1.5 Basic commands

Table shows the LoadLeveler commands, which give administrators and users access to LoadLeveler. These commands are located in the LoadLeveler release directory /usr/lpp/LoadL/full/bin.

Table 1-2  Summary of LoadLeveler commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
<th>Admin</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>llacctmrg</td>
<td>Merges all machine history files into single file.</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>llcancel</td>
<td>Cancels a job.</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>llclass</td>
<td>Displays information on available classes.</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>llctl</td>
<td>Controls daemons locally or globally.</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>llcegrpmaint</td>
<td>Maintains DCE group and principal names.</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>llextSDR</td>
<td>Extracts node adapter information from SDR.</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>llfavorjob</td>
<td>Modifies the priority of a job.</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>llhold</td>
<td>Holds or releases a job.</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>llinit</td>
<td>Initializes a machine into the cluster.</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>llmatrix</td>
<td>Displays GANG matrix information.</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>llmodify</td>
<td>Modifies job step.</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>llpreempt</td>
<td>Preempts/Unpreempts a job step.</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>llprio</td>
<td>Modifies user priority of a job step.</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>llq</td>
<td>Queries the job queue.</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>llstatus</td>
<td>Displays machine status.</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>llsubmit</td>
<td>Submits a job.</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>llsummary</td>
<td>Displays completed job resource information.</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

1.6 Graphical user interface (GUI)

In addition to interacting with LoadLeveler via the command line, users can also use one of the two versions of the LoadLeveler GUI.

xloadl is used on machines in the cluster that run some or all of the LoadLeveler daemons and gives access to the full LoadLeveler functionality.
xloadl.so is used on submit-only machines. These machines sit outside the cluster and do not run LoadLeveler daemons. They only have a limited command set available, allowing users to submit jobs and query machine and job information.

The GUI's functionality can be customized to a site's requirements. This is discussed in detail in Section 9.2.1, “Customizing the LoadLeveler GUI” on page 147.

### 1.7 Application programming interfaces (APIs)

The information used within LoadLeveler (and several means to control the behavior of LoadLeveler) are accessible to users via a number of APIs, which can be called from C or C++. The header file llapi.h in the directory /usr/lpp/LoadL/full/include defines all the data structures and routines necessary to use the subroutines located in the shared library libllapi.a in the directory /usr/lpp/LoadL/full/lib.

The following APIs are available:

- Accounting
- Checkpointing
- Submit
- Data Access
- Parallel Job
- Workload Management
- Query
- User Exits

Several examples of using the APIs, such as the use of external scheduler and common gateway interface (CGI) scripts to generate Web status pages, are discussed in detail in Chapter 8, “Customized schedulers using API” on page 115.
1.8 Simple examples of job command files

Job submission to LoadLeveler is accomplished using a job command file. The job command file is a shell script containing keywords embedded in comments beginning with # @. These keywords inform LoadLeveler of the job’s environment, the program to execute, where to write output files, whether the job is serial or parallel, and the resources required for the job to run.

Example 1-2 and Example 1-3 show the general layout of command files. An explanation and listing of the keywords can be found in the manual *Using and Administering*, SA22-7311.

**Example 1-2  Job command file for a serial job**

```plaintext
#!/bin/ksh
# @ job_type = serial
# @ class = serial_s
# @ initialdir = /u/markr
# @ executable = a.out
# @ output = test.$(jobid).$(stepid).out
# @ error = test.$(jobid).$(stepid).err
# @ notification = always
# @ notify_user = markr@sp4en0.msc.itso.ibm.com
# @ requirements = (Arch == "R6000") && (OpSys == "AIX51")
# @ queue
```

In Example 1-2, we see that all the LoadLeveler directives are contained within shell script comments, using the # @ trigger string. Here we specify that the job is serial, and is to run in the serial_s job class. The initial directory is the home directory of the user markr, and the executable to be run, along with the location of the files for standard output and standard error, relative to the initial directory. markr will receive e-mail notification when the job starts, if it fails, and when it completes. The job is required to run on an RS/6000 running AIX 5.1. # @ queue is the directive that triggers the execution of the job step.

**Example 1-3  Job command file for a parallel job**

```plaintext
#!/bin/ksh
# @ job_type = parallel
# @ class = poe_l
# @ initialdir = /u/markr/MIX3D
# @ executable = /usr/bin/poe
# @ arguments = ./mix3d
# @ input = Data/1n16p-256
# @ output = MIX3D.$(jobid).$(stepid).out
# @ error = MIX3D.$(jobid).$(stepid).out
# @ notification = never
# @ node = 2
# @ tasks_per_node = 4
```
Some of the directives in Example 1-3 on page 7 are similar to those in Example 1-2 on page 7. However, this time we ran a parallel job. The executable is the /usr/bin/poe command, which runs a parallel executable ./mix3d in the initial directory. In this example, data comes from a data file, which is located in a subdirectory of the initial directory. The user markr will not receive any e-mail notification this time. The other key words specify the requirements of the job: it is to use two nodes, with four tasks on each node, making eight tasks all together. The program uses the MPI message passing library, and uses User Space to communicate over the switch. The environment variable MP_SHARED_MEMORY is set to yes to indicate that when MPI passes messages within a node, communication is done via a shared memory segment, rather than using the switch adapter. This is discussed in Section 5.5.3, “Using shared memory” on page 80.
Easy installation

This chapter describes the quick installation method for the LoadLeveler product and the basic configurations in an SP cluster. More advanced configurations are discussed in the later chapters. The following topics are covered in this chapter:

- LoadLeveler product installation on nodes in the cluster
- LoadLeveler user setup
- Creating the /var/loadl file system
- Running the LoadLeveler installation script
- Creating basic configuration file
- Creating an adapter stanza for the SP switch
- Starting LoadLeveler
2.1 Quick steps to install

In this chapter, we provide quick installation and configuration guidelines for setting up a simple LoadLeveler environment, with minimal customization. For simplicity, we assume that the installation is to be performed on an SP rather than a general cluster of machines.

2.1.1 Install the product on all the nodes

The first step for configuring LoadLeveler is to install the product on all the SP nodes. To do this, we have to copy the install images from the product media to the control workstation. We created the /spdata/sys1/install/pssplpp/LPP directory as a staging directory for all the LPP products. We copied the LoadLeveler filesets to this directory from the installation media, using the command in Example 2-1.

Example 2-1 Copy the installation files to disk

```
# bffcreate -d /dev/cd0 -f all -t /spdata/sys1/install/pssplpp/LPP -X
```

In order to install LoadLeveler on a single node, we log on to the node and mount this directory using the command shown in Example 2-2. Note that in a standard SP installation, the /spdata/sys1/install/pssplpp directory already has been NFS-exported.

Example 2-2 Mount the installation directory

```
# mount sp6en0:/spdata/sys1/install/pssplpp /mnt
```

The next step is to install the product using the `installp` command (Example 2-3).

Example 2-3 Install the LoadLeveler filesets

```
# installp -acgXd/mnt/LPP LoadL
```

Verify the installation of the product using the `lslpp` command. The output should be similar to Example 2-4.

Example 2-4 Verify the installation

```
# lslpp -l |grep LoadL
  LoadL.full      3.1.0.0 COMMITTED LoadLeveler
  LoadL.html     3.1.0.0 COMMITTED LoadLeveler HTML Pages
  LoadL.msg.En_US 3.1.0.0 COMMITTED LoadLeveler Messages - U.S.
  LoadL.msg.en_US 3.1.0.0 COMMITTED LoadLeveler Messages - U.S.
  LoadL.pdf      3.1.0.0 COMMITTED LoadLeveler PDF Documentation
  LoadL.so       3.1.0.0 COMMITTED LoadLeveler (Submit only)
  LoadL.tguides  3.1.0.0 COMMITTED LoadLeveler TaskGuides
  LoadL.full     3.1.0.0 COMMITTED LoadLeveler
```

10 Workload Management with LoadLeveler
In order to install LoadLeveler on all the nodes of an SP, the commands shown above can be issued on multiple nodes using the dsh (distributed shell) command.

### 2.1.2 Set up the loadl user

It is usual (but not essential) to create a group called loadl and a user also called loadl, which belongs to the loadl group, to act as the LoadLeveler administrator. We create this user, as shown in Example 2-5, on the SP control workstation and we then use the supper command to replicate the user definition onto the nodes. Note that these commands assume both User Management and File Collection Management have been enabled on the SP.

**Example 2-5  Create the Administrative user**

```
mkgroup -a loadl
spmkuser pgrp=loadl groups=loadl home=sp6en0:/home/loadl loadl
dsh -av /var/sysman/supper update user.admin
```

### 2.1.3 Create /var/loadl file system

Each node in the cluster requires space for the local spool, execute, and log directories. Although these are referred to as local, it is only necessary for them to be unique. However, for performance reasons, we strongly recommend that they be physically local to the node. For this reason, we create a separate JFS file system /var/loadl on each node, as shown in Example 2-6.

**Example 2-6  Create the local file system**

```
# smitty crjfsstd
Add a Standard Journaled File System

Type or select values in entry fields.
Press Enter AFTER making all desired changes.

[Entry Fields]

Volume group name                        rootvg
* SIZE of file system (in 512-byte blocks)  [32768]
#
* MOUNT POINT
Mount AUTOMATICALLY at system restart?    yes +
PERMISSIONS                                read/write +
Mount OPTIONS                              [] +
Start Disk Accounting?                    no +
Fragment Size (bytes)                      4096 +
Number of bytes per inode                 4096 +
```

Chapter 2. Easy installation  11
Once the file system has been created, it should be mounted, as shown in Example 2-7.

**Example 2-7  Mount the local file system**

```
# mount /var/loadl
```

### 2.1.4 Change the ownership

The next step is to change the ownership of the /var/loadl file system to the loadl user:

```
# mount /var/loadl
# chown -R loadl.loadl /var/loadl
```

The commands used for setting up /var/loadl should be executed on all nodes of the cluster using `dsh`. The command used to create the /var/loadl file system can be extracted from the tail of the smit.script in the root user’s home directory.

### 2.1.5 Run the LoadLeveler installation script

The LoadLeveler installation script `llinit` should be run on each node as the loadl user. We made the choices shown in Table 2-1 for the LoadLeveler configuration.

**Table 2-1  LoadLeveler initial setup**

<table>
<thead>
<tr>
<th>Component</th>
<th>File or directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local directory</td>
<td>/var/loadl</td>
</tr>
<tr>
<td>Release directory</td>
<td>/usr/lpp/LoadL/full</td>
</tr>
<tr>
<td>Central Manager</td>
<td>sp6n01.msc.itso.ibm.com</td>
</tr>
</tbody>
</table>

We ran the command in Example 2-8 as a loadl user to execute the initial script.

**Example 2-8  Initialize LoadLeveler**

```
$ cd /usr/lpp/LoadL/full/bin
$ ./llinit -local /var/loadl -release /usr/lpp/LoadL/full -cm sp6n01.msc.itso.ibm.com
```

The `llinit` script does following:

- Creates the files `LoadL_config` and `LoadL_admin` in the home directory
Chapter 2. Easy installation

2.1.6 Create links for config file

By default, the llimit program creates a file, /var/loadL.LoadL_config.local, local to each node, in order to enable node-specific customization. However, because these files are physically distributed around the nodes, they are inconvenient to administer.

Although we recommend that the spool, execute, and log directories be physically local on the nodes, it is only necessary that the local configuration file be distinguishable on the individual nodes, and not physically local. Because this file is only read periodically, performance is not likely to be an issue.

We recommend that the local configuration file be stored in a subdirectory of the loadl user’s home directory, which we have assumed is NFS-mounted across the nodes. Example 2-9 shows how we set this up with a scheme of symbolic links.

Example 2-9  Create local configuration files

$ cd /u/loadl
$ mkdir LOCAL
$ cd LOCAL
$ cp /var/loadl/LoadL_config.local .
$ ln -s LoadL_config.local LoadL_config.sp6n11
$ ln -s LoadL_config.local LoadL_config.sp6n12

In the global configuration file, /etc/LoadL.cfg, we changed the LOCAL_CONFIG line to read:

LOCAL_CONFIG = $(tilde)/LOCAL/LoadL_config.$(host)

On node sp6n11, for example, LOCAL_CONFIG evaluates to /u/loadl/LOCAL/LoadL_config.sp6n11, which is then a symbolic link to a common local configuration file. Using this method, similar nodes can have local configuration files which are symbolic links to the same file in /u/loadl/LOCAL.
2.1.7 Execute llextSDR

The llextSDR command extracts machine and adapter information from the System Data Repository (SDR) for the SP configuration. The output of this command provides the details about the machine and adapter configuration of the nodes. Run this command and redirect this output into a temporary file. Then include this file in the LoadL_admin file under the machine stanza section.

Example 2-10 shows the output from this command, executed as the loadl user.

Example 2-10  Run the llextSDR command

```
$ llextSDR > /u/loadl/llextSDR.out
$ cat llextSDR.out
#llextSDR: System Partition = "sp6en0" on Mon May 21 13:30:08 2001
sp6n14: type = machine
  adapter_stanzas = sp6sw14 sp6n14
  spacct_exclude_enable = false
  alias = sp6sw14
sp6sw14: type = adapter
  adapter_name = css0
  network_type = switch
  interface_address = 192.168.16.14
  interface_name = sp6sw14
  switch_node_number = 13
  css_type = SP_Switch_Adapter
sp6n14: type = adapter
  adapter_name = en0
  network_type = ethernet
  interface_address = 192.168.6.14
  interface_name = sp6n14
```

2.1.8 Edit LoadL_admin

Merge the machine stanza for the central manager produced by the llinit script, together with the output of the llextSDR command. In addition, we also need to add class information to the LoadL_admin file. The default LoadL_admin file has a class inter_class defined, and this is set by the default class for interactive parallel jobs. Add other class stanzas as required, based on the examples given in comments in the LoadL_admin file.
2.1.9 Edit LoadL_config

Next, edit the LoadL_config file. We recommend that root be added as a LoadLeveler administrator. This enables root to start LoadLeveler from one of the system startup scripts. We change LOCAL_CONFIG to point to the local configuration files, as described in Section 2.1.6, “Create links for config file” on page 13. To enable any classes in the previous step, we can add a class statement to LoadL_config:

```
Class = inter_class(2)
```

2.1.10 Start LoadLeveler

Ensure that the directory $HOME/bin is in the PATH for the LoadLeveler administrator. Then issue the following command to start LoadLeveler on all nodes in the cluster:

```
llctl -g start
```

This assumes that the loadl user is able to issue the rsh command to all nodes in the cluster. This can be enabled by adding nodes to the file "$HOME/.rhosts". Alternatively, the root user can enable this an a system-wide basis by editing the file “/etc/hosts.equiv”. A third alternative in an SP environment is to set up a Kerberos principal for the LoadLeveler administrator, so that loadl can use kerberos-authenticated remote commands.

Once LoadLeveler has been started, the status of the cluster can be verified using the llstatus command. We configured for scheduling function, so the schedd is running on one system and not on all the nodes. A sample output is given in Example 2-11.

```
Example 2-11  Output of llct1 command

<table>
<thead>
<tr>
<th>Name</th>
<th>Schedd</th>
<th>InQ</th>
<th>Act</th>
<th>Startd</th>
<th>Run</th>
<th>LdAvg</th>
<th>Idle</th>
<th>Arch</th>
<th>OpSys</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp6en0.msc.itso.ibm.com</td>
<td>Avail</td>
<td>0</td>
<td>0</td>
<td>Down</td>
<td>0</td>
<td>0.04</td>
<td>1</td>
<td>R6000</td>
<td>AIX51</td>
</tr>
<tr>
<td>sp6n01.msc.itso.ibm.com</td>
<td>Down</td>
<td>0</td>
<td>0</td>
<td>Idle</td>
<td>0</td>
<td>0.09</td>
<td>9999</td>
<td>R6000</td>
<td>AIX51</td>
</tr>
<tr>
<td>sp6n03.msc.itso.ibm.com</td>
<td>Down</td>
<td>0</td>
<td>0</td>
<td>Idle</td>
<td>0</td>
<td>0.07</td>
<td>9999</td>
<td>R6000</td>
<td>AIX51</td>
</tr>
<tr>
<td>sp6n09.msc.itso.ibm.com</td>
<td>Down</td>
<td>0</td>
<td>0</td>
<td>Idle</td>
<td>0</td>
<td>1.04</td>
<td>9999</td>
<td>R6000</td>
<td>AIX51</td>
</tr>
<tr>
<td>sp6n10.msc.itso.ibm.com</td>
<td>Down</td>
<td>0</td>
<td>0</td>
<td>Idle</td>
<td>0</td>
<td>0.15</td>
<td>9999</td>
<td>R6000</td>
<td>AIX51</td>
</tr>
<tr>
<td>sp6n11.msc.itso.ibm.com</td>
<td>Down</td>
<td>0</td>
<td>0</td>
<td>Idle</td>
<td>0</td>
<td>1.37</td>
<td>9999</td>
<td>R6000</td>
<td>AIX51</td>
</tr>
<tr>
<td>sp6n12.msc.itso.ibm.com</td>
<td>Down</td>
<td>0</td>
<td>0</td>
<td>Idle</td>
<td>0</td>
<td>1.00</td>
<td>9999</td>
<td>R6000</td>
<td>AIX51</td>
</tr>
<tr>
<td>sp6n13.msc.itso.ibm.com</td>
<td>Down</td>
<td>0</td>
<td>0</td>
<td>Idle</td>
<td>0</td>
<td>1.01</td>
<td>9999</td>
<td>R6000</td>
<td>AIX51</td>
</tr>
<tr>
<td>sp6n14.msc.itso.ibm.com</td>
<td>Down</td>
<td>0</td>
<td>0</td>
<td>Idle</td>
<td>0</td>
<td>1.06</td>
<td>9999</td>
<td>R6000</td>
<td>AIX51</td>
</tr>
</tbody>
</table>

R6000/AIX51                   | 9 machines | 0 | jobs | 0 | running |
Total Machines                | 9 machines | 0 | jobs | 0 | running |

The Central Manager is defined on sp6en0.msc.itso.ibm.com
The BACKFILL scheduler is in use

All machines on the machine_list are present.
New features

In this chapter, we briefly describe the new features that have been added to LoadLeveler after LoadLeveler Version 2 Release 1. The following topics will be covered:

- Task assignment
- Consumable resources and WLM integration
- GSmonitor
- Process tracking
- DCE support
- 64-bit support
- GANG scheduling
- Preemption of jobs
- Checkpoint/Restart of jobs in LoadLeveler
- Adapter striping
3.1 Task assignment

LoadLeveler was originally developed before IBM introduced SMP machines to the RS/6000 range. Therefore, all nodes in a LoadLeveler pool had a single CPU. For batch parallel jobs, it made no sense to run more than one parallel task on a single node. Before LoadLeveler Version 2.2, the only way to influence the mapping of a task to a particular node was to designate a node as a master node by using the master_node_exclusive keyword in the machine stanza of the LoadL_admin file. A class could then be specified with master_node_requirement = true, so that the first task of a parallel job, which in many parallel applications is the task performing I/O, maps to the machine that sets master_node_exclusive = true, which may have locally attached storage.

It should be noted that an interactive parallel job, which uses a host file by calling POE with the -hfile hostfile flag, or exporting the environment variable MP_HOSTFILE=hostfile, has complete control over mapping tasks to nodes.

More recently, RS/6000 nodes with multiple CPUs have become available, which require greater flexibility to map tasks to nodes and CPUs. LoadLeveler Version 2.2 introduced a number of keywords to control task assignment. The use of total_tasks, tasks_per_node, and blocking is described in detail in the Using and Administering, SA22-7311 and also in Section 5.1, “Task assignment” on page 58. For more arbitrary mapping of tasks to nodes, the task_geometry keyword was made available. This allowed an expression such as

\[ \# @ task\_geometry = {{(5,2),(1,3),(4,6,0)}} \]

to be specified for a job step. However, all tasks and nodes must be accounted for in a task\_geometry expression.

3.2 Consumable resources and WLM integration

Before Version 2.2, LoadLeveler had no knowledge of multithreading tasks on an SMP node. LoadLeveler expected a single task on an SMP node to consume a single CPU. However, a single task or process could be multithreaded so that, in practice, multiple CPUs were consumed. So, for example, a four-way SMP node could have four tasks scheduled to run by LoadLeveler, but each task could be attempting to run threads on the CPUs allocated to other tasks.

In a similar manner, LoadLeveler had no mechanism beyond enforcing the standard AIX process limits to intelligently manage the consumption of real and virtual memory by a job. Users found that two memory-hungry applications running together on the same SMP node could cause excessive amounts of paging.
LoadLeveler Version 2.2 introduced the concept of *consumable resources*, where a node could be specified to have a certain quantity of resources available, such as CPUs, and real and virtual memory. Jobs could then specify, either explicitly in the job command file, or implicitly by means of defaults specified for the job class in the LoadL_admin file, what resources they would consume.

In this manner, a parallel job that mixed the use of MPI and OpenMPI could run on, say, two 16-way nodes, as eight tasks, each of which consists of four threads, by specifying:

```plaintext
total_tasks = 8
tasks_per_node = 4
resources = ConsumableCpus(4)
```

in the job command file.

The concept of ConsumableCpus, ConsumableMemory, and ConsumableVirtualMemory are built into LoadLeveler and further custom consumable resources may be defined. In addition, floating resources, which are fixed in total number, but not tied to a particular node, can be defined. These can be used for mimicking floating licenses or for managing a fixed floating resource, such as a network-attached tape device.

One drawback of the implementation of consumable resources with LoadLeveler Version 2.2 is that there is no mechanism of enforcement available, so that users could state that they consume four CPUs, while at the same time running eight threads. AIX Version 4.3.3 introduced the Workload Management System (WLM), which has been further enhanced in AIX 5L. WLM is integrated into AIX, and allows the use of resources to be strictly controlled. At LoadLeveler Version 2.2, WLM can be used to enforce resource usage, but only by statically defining WLM classes to match the LoadLeveler classes.

LoadLeveler Version 3.1 now uses AIX 5L's WLM to enforce the use of ConsumableCpus and ConsumableMemory. WLM integration is enabled by setting the keywords `ENFORCE_RESOURCE_SUBMISSION = TRUE`, and `ENFORCE_RESOURCE_USAGE` to a list of the consumable resources that should be enforced.

The use of WLM to enforce CPU and real memory usage is entirely transparent to the administrator, and requires no WLM configuration. However, it should be noted that when LoadLeveler uses WLM, all previous WLM configuration is overwritten.
3.3 gsmonitor daemon

By default, machines in the LoadLeveler pool send a heartbeat to the negotiator daemon on the Central Manager every polling interval. If no heartbeat is received after two polling interval periods, then the Central Manager marks that machine as down.

Group services (GS) is a distributed subsystem of the Parallel System Support Programs on RS/6000 SP. It is one of the several subsystems in Parallel System Support Programs that provide a set of high availability services.

Where the group services subsystem is available, such as on an RS/6000 SP running Parallel System Support Programs, or a cluster of @server pSeries machines, then this function can be provided quickly and reliably. LoadLeveler Version 2.2 added the gsmonitor daemon (LoadL_GSmonitor), which is able to notify the negotiator daemon when a machine is no longer reachable.

The gsmonitor is enabled by setting GSMONITOR_RUNS_HERE = TRUE in the local configuration file of one or more nodes in the LoadLeveler pool.

3.4 Process tracking

Since a LoadLeveler job may be an arbitrary shell script, it is possible for a job to spawn child processes. Prior to LoadLeveler Version 2.2, it was possible for some of these child processes to continue to exist after the parent job had exited or had been cancelled. These child processes could possibly continue to consume resources, which could degrade performance and interfere with the normal scheduling of jobs by LoadLeveler.

LoadLeveler Version 2.2 added a kernel extension, which is able to keep track of all processes throughout the LoadLeveler cluster created by each job. When a job exits or is terminated, LoadLeveler is then able to terminate all these child processes reliably.

Process tracking is enabled by setting PROCESS_TRACKING = TRUE in the global configuration file. The default location for the process tracking kernel extension LoadL_pt_ke is $HOME/bin. This may be optionally overridden by specifying an alternate directory with the keyword PROCESS_TRACKING_EXTENSION.
3.5 Distributed computing environment (DCE)

Full support for distributed computing environment (DCE) security features was introduced with Parallel System Support Programs Version 3.2 and the corresponding release of LoadLeveler Version 2.2. LoadLeveler's interactions with DCE are carried out using PSSP's security services API. LoadLeveler is able to use DCE to authenticate the identity of users interacting with LoadLeveler, and to authorize users to use LoadLeveler services. The LoadL_starter daemon is able to grant the same DCE credentials to a running job as are owned by the submitting user.

A command, `lldcegrpmaint`, was added to automate the setup of LoadLeveler DCE integration. It creates the LoadL-admin and LoadL-services DCE groups, and adds DCE principals for the LoadLeveler daemons to the LoadL-services group.

3.6 64-bit support

AIX has provided support for 64-bit applications since AIX Version 4.3 was introduced. However, programs written using the MPI libraries provided by the Parallel Processing Environment fileset ppe.poe have been restricted to using 32-bit data types when communicating using MPI calls. Similarly, LoadLeveler up to and including Version 2.2 has only provided support for 32-bit applications.

LoadLeveler Version 3.1, together with Parallel System Support Programs Version 3.4 and Parallel Processing Environment Version 3.2 fully supports 64-bit applications. The support is largely transparent to users and administrators. The main externally visible changes are that 64-bit values may be specified where appropriate, for example when specifying process limits. And display commands, such as `llq` and `llstatus`, have been enhanced to display 64-bit data.

3.7 GANG scheduling

LoadLeveler Version 2.2 allows multiple jobs to execute simultaneously on a group of nodes, and also allows multiple jobs to timeshare on a single node. There are two main limitations that must be considered when tasks of different jobs are running simultaneously on the same node.
Firstly, parallel programs that use the User Space (US) libraries to communicate over the SP switch use User Space Windows. A US Window is an area of memory on the switch adapter used by the Communications Subsystem (CSS) to hold messages in transit. US Windows are a scarce resource - the initial SP switch and switch adapter implementations had a single US Window. A process using US owned the window for its lifetime, so that use of the Switch in US was effectively dedicated to a single job. More recent switch/adapter implementations have increased the number of US windows. For example, Power3 thin/wide nodes attached to the SP_Switch (or TBS switch) allow four US windows, and Power3 high nodes attached to the SP_Switch2 (or Colony switch) allow sixteen US windows. Nevertheless, before PSSP 3.4, a US window was owned by the parallel process for the lifetime of the process.

Second, although parallel tasks can timeshare on nodes, there has been no mechanism to coordinate the context switching between tasks, leading to inefficiency as parallel tasks arrive at synchronization points at different times.

The enhanced CSS software allows a US task to relinquish ownership of a US window, which may then be taken over by another task belonging to another job. LoadLeveler 3.1 exploits this enhancement to allow the tasks of multiple parallel jobs to share nodes with the switching of resources, including US windows, between the tasks coordinated across the system.

LoadLeveler 3.1 introduces a third internal scheduler, known as the GANG Scheduler, implementing this time-sharing feature. The GANG Scheduler uses the concept of a GANG Matrix (which may be viewed using the `llmatrix` command). The GANG Matrix is a two-dimensional matrix where columns represent processors, and rows represent time slices. A simple example of a GANG Matrix is shown in Table 3-1.

<table>
<thead>
<tr>
<th>Slice 1</th>
<th>Node 1</th>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 2</th>
<th>Node 3</th>
<th>Node 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU 1</td>
<td>CPU 2</td>
<td>CPU 1</td>
<td>CPU 2</td>
<td>CPU 1</td>
<td>CPU 2</td>
</tr>
<tr>
<td>Slice 1</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>Slice 2</td>
<td>B</td>
<td>B</td>
<td>D</td>
<td>D</td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>Slice 3</td>
<td>A</td>
<td>A</td>
<td>E</td>
<td>E</td>
<td>A</td>
<td>E</td>
</tr>
<tr>
<td>Slice 4</td>
<td>B</td>
<td>B</td>
<td>C</td>
<td>C</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>Slice 5</td>
<td>A</td>
<td>A</td>
<td>D</td>
<td>D</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>Slice 6</td>
<td>B</td>
<td>B</td>
<td>E</td>
<td>E</td>
<td>B</td>
<td>E</td>
</tr>
</tbody>
</table>
In this example, five jobs are active on three nodes, each of which has two CPUs. Node 1 has two jobs, A and B, with a multiprogramming level of 2, and Nodes 2 and 3 have three jobs, C, D, and E, with a multiprogramming level of 3. At the end of each time slice, all the processors simultaneously switch contexts between the jobs, including the US windows.

There are a number of new keywords in the LoadL_admin file which control the GANG Scheduler. The most fundamental is GANG_MATRIX_TIME_SLICE. The minimum value for this parameter is 60 seconds.

One of the main anticipated uses of the GANG Scheduler is for running applications which may fail early in their execution. In this scenario, a number of applications may be running simultaneously in the system, and those that fail drop out of the system early. This is better than running the applications sequentially, where an individual job may be queued for a long time before beginning execution and then failing almost immediately. Similarly, if multiple jobs are executing at the same time, the job output could be used to determine if jobs are making satisfactory progress, and errant jobs can be terminated.

### 3.8 Preemption

The enhancement to the CSS (described in Section 3.7, “GANG scheduling” on page 21), allowing jobs to context switch, has enabled LoadLeveler Version 3.1 to implement a preemption scheme. In many situations it is required to interrupt lower priority work in order to allow a higher priority job to begin execution immediately. An example is weather forecasting, where research and development jobs may fill up a system, but the operational weather forecast has to be run at a particular time of day.

A job may be preempted manually using the `llpreempt` command. In addition, preemption can happen automatically, based on job classes. For example, a site may specify:

```
PREEMPT_CLASS[Forecast] = ALL { Research Development }
```

This would allow a job submitted to class Forecast to preempt jobs running in Research and Development classes. ALL specifies that jobs in the two specified classes are not allowed to timeshare using GANG scheduling on the same node as a running Forecast job. If a preemtping job is allowed to timeshare with the lower priority jobs, then the keyword ENOUGH may be used and only ENOUGH low priority jobs to allow this job to run will be preempted.
3.9 Checkpoint/restart

The ability to take a checkpoint of an executing serial job, and to restart the job from a checkpoint, possibly on a different node, has been available in LoadLeveler since the earliest versions. A checkpoint file contains the program's data segment, stack, heap, register contents, signal states, and the states of the open files at the time of the checkpoint. As such a checkpoint file is potentially very large.

Checkpoints may either be “user initiated”, where the application makes a call to the ckpt() routine, or system initiated, where LoadLeveler automatically takes a checkpoint at specified intervals. A job which is vacated, or which is running on a machine which fails, may then be restarted on a new machine, using the most recent checkpoint taken before the job failed.

Before LoadLeveler Version 2.2, only serial jobs could be checkpointed and restarted. LoadLeveler Version 2.2 introduced checkpointing for parallel jobs, although only user-initiated checkpointing was available. Various keywords and environment variables are available to control when and how the checkpoints are taken, and where the checkpoint files are stored. In AIX 5L the bos.cpr fileset provides the Checkpoint/Restart Runtime and Tools. LoadLeveler Version 3.1 is able to exploit this enhanced support from AIX to provide a more general checkpoint /restart facility with fewer restrictions and greater reliability.

Checkpoint/restart is now supported for batch serial jobs, and for both interactive and batch parallel jobs. Given that checkpoint files are likely to be large for all but the smallest jobs, and that all tasks of a parallel job will generate a checkpoint file, the facility is only likely to be useful in practice when used in conjunction with a high-performance file system, such as GPFS.

3.10 Adapter striping

SP systems based on 375_MHz_POWER3_SMP_High nodes (NightHawk2) and SP_Switch2 (Colony) may use multiple switch adapters to increase communications bandwidth. Parallel jobs may use multiple paths between pairs of nodes using either IP or US.

IP striping involves setting up a multi-link interface, ml0, and corresponding IP address in the LoadL_admin file, which may then be used in a job command file. In the adapter stanza of the LoadL_admin file, the administrator specifies:

```
multilink_address = a.b.c.d
multilink_list = css0,css1
```
Next, in the job command file, the user specifies:

```plaintext
network.MPI = css,IP
```

US striping is invoked by using the virtual device css in a network statement in the job command file

```plaintext
network.CPI = css,US
```

The css device is generated automatically, and does not need to be specified in the LoadL_admin file.
Managing jobs

This chapter describes a number of ways in which jobs are managed in a LoadLeveler environment. The following topics are covered:

- Job steps
- Job filters
- Run-time environment variables
- Managing job environment
- Using job command file variables
- Managing classes
- Managing limits
- Managing job queues
- Managing job execution
4.1 Job steps

Although we talk about jobs, the basic unit of work dispatched by LoadLeveler is the **job step**. A job step is signalled in a user's job command file by the `@ queue` directive, and a single job contained within a job command file may consist of multiple job steps. Consider Example 4-1:

**Example 4-1 Simple two-step job**

```
# @ job_name = test1
# @ step_name = step_1
# @ job_type = serial
# @ initialdir = /u/peterm/LL_RESIDENCY/TESTS/TEST7
# @ input = /dev/null
# @ output = ${job_name}.${jobid}.${stepid}.out
# @ error = ${job_name}.${jobid}.${stepid}.err
# @ class = express
# @ notification = never
# @ queue
# @ step_name = step_2
# @ queue
hostname
sleep 60
```

The two job steps shown in Example 4-1 are independent, and if nodes are available, they can run simultaneously, as show in Example 4-2.

Although this example is trivial, simple derivations can still be useful in practice. Suppose that the same program needs to be run against multiple input files. If these files are called prog.dat.0, prog.dat.1,prog.dat.2,... then the job command file can simply say

```
# @ executable = prog.exe
# @ input = prog.dat.$(stepid)
# @ queue
# @ queue
# @ queue
```

to achieve the desired effect.

**Example 4-2 Output of Example 4-1**

```
ode09 (504)% llsubmit test1.cmd
llsubmit: The job "node09.stc.uk.ibm.com.480" with 2 job steps has been submitted.
node09 (505)% llq
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>node09.480.0</td>
<td>peterm</td>
<td>6/6 00:42 R</td>
<td>50</td>
<td>express</td>
<td>node16</td>
<td></td>
</tr>
<tr>
<td>node09.480.1</td>
<td>peterm</td>
<td>6/6 00:42 R</td>
<td>50</td>
<td>express</td>
<td>node15</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 4. Managing jobs

2 job steps in queue, 0 waiting, 0 pending, 2 running, 0 held
node09 (506)% llq

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>node09.480.0</td>
<td>peterm</td>
<td>6/6</td>
<td>00:42</td>
<td>C</td>
<td>50</td>
<td>express</td>
</tr>
<tr>
<td>node09.480.1</td>
<td>peterm</td>
<td>6/6</td>
<td>00:42</td>
<td>C</td>
<td>50</td>
<td>express</td>
</tr>
</tbody>
</table>

0 job steps in queue, 0 waiting, 0 pending, 0 running, 0 held

node09 (509)% cat test1.480.0.out
node16.stc.uk.ibm.com
node09 (510)% cat test1.480.1.out
node15.stc.uk.ibm.com

It is important to note that the body of the job command file is executed for each job step. So a job command file containing

```bash
# @ step_name = step_1
# @ queue
echo "I am step 1"
# @ step_name = step_2
# @ queue
echo "I am step 2"
```

does not produce two files containing “I am step x.” Each output file contains the two lines echoed. Instead, the body of the script needs to determine which step it is executing:

```bash
# @ step_name = step_1
# @ queue
# @ step_name = step_2
# @ queue
case $LOADL_STEP_NAME in
  step_1)
    echo "I am step 1"
    ;;
  step_2)
    echo "I am step 2"
    ;;
esac
```

One of the most common uses of multi-step jobs is where a later step depends on results of an earlier step, as in Example 4-3.

Example 4-3 Job step dependencies

```bash
# @ step_name = step_1
# @ queue
# @ step_name = step_2
```
# @ dependency = ( step_1 == 0 )
# @ queue
# @ step_name = step_3
# @ dependency = ( step_1 != 0 )
# @ queue
# @ step_name = step_4
# @ dependency = ( step_2 == CC_NOTRUN )
# @ queue
case $LOADL_STEP_NAME in
  step_1)
    echo "I am failing in step 1"
    exit 1
  ;;
  step_2)
    echo "I am step 2, step 1 was fine"
  ;;
  step_3)
    echo "I am step 3, step 1 bought the farm"
  ;;
  step_4)
    echo "I am step 4. Why didn't step 2 run?"
  ;;
esac

Note the use of the value CC_NOTRUN in the dependency for step_4. An alternative pre-defined symbolic value is CC_REMOVED, which is the exit code set when a step is removed.

Example 4-3 produces the output shown in Example 4-4.

**Example 4-4  Output from Example 4-3**

```bash
node9 (513)% llsubmit test4.cmd
llsubmit: The job "node9.stc.uk.ibm.com.486" with 4 job steps has been submitted.

node9 (514)% llq

```

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>node9.486.0</td>
<td>peterm</td>
<td>6/6 01:14 R 50</td>
<td>express</td>
<td>node15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>node9.486.1</td>
<td>peterm</td>
<td>6/6 01:14 NQ 50</td>
<td>express</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>node9.486.2</td>
<td>peterm</td>
<td>6/6 01:14 NQ 50</td>
<td>express</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>node9.486.3</td>
<td>peterm</td>
<td>6/6 01:14 NQ 50</td>
<td>express</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4 job steps in queue, 0 waiting, 0 pending, 1 running, 3 held

```bash
node9 (515)% llq

```

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>node9.486.2</td>
<td>peterm</td>
<td>6/6 01:14 R 50</td>
<td>express</td>
<td>node13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>node9.486.3</td>
<td>peterm</td>
<td>6/6 01:14 R 50</td>
<td>express</td>
<td>node15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

30  Workload Management with LoadLeveler
We note several things in the output:

- When step_1 is running, all the other steps are in the NQ or NotQueued job state. LoadLeveler knows that their dependencies are not satisfied, and, therefore, they are not being considered for dispatch by the scheduler.

- When step_1 has failed, step_2 is set to NR or NotRun and step_4 is then run.

- At the same time, step_3 can be run, as its dependency is satisfied.

- Since step_3 and step_4 are independent, they can run simultaneously on different nodes.

If a particular job step requires the same node as a previous job step, a variant of the requirements directive can be used (see Example 4-5).

### Example 4-5  Multiple steps on the same node

```bash
# @ step_name = step_1
# @ queue
# @ step_name = step_2
# @ dependency = ( step_1 == 0 )
# @ requirements = ( Machine == machine.step_1 )
# @ queue
case $LOADL_STEP_NAME in
  step_1)
    echo "I am step 1, running on $(hostname)" | tee /tmp/test5.dat
    ;;
```
Example 4-5 produces the output shown in Example 4-6.

**Example 4-5 Output from Example 4-5**

```bash
echo "I am step 1, running on $(hostname)"
echo "Where did step 1 run?"
cat /tmp/test5.dat
;;;
esac
```

Example 4-6

```bash
node09 (528)% llsubmit test5.cmd
llsubmit: The job "node09.stc.uk.ibm.com.492" with 2 job steps has been submitted.
node09 (529)% llq
<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>node09.492.0</td>
<td>peterm</td>
<td>6/6</td>
<td>01:37</td>
<td>R 50</td>
<td>express</td>
<td>node13</td>
</tr>
<tr>
<td>node09.492.1</td>
<td>peterm</td>
<td>6/6</td>
<td>01:37</td>
<td>NQ 50</td>
<td>express</td>
<td></td>
</tr>
</tbody>
</table>

2 job steps in queue, 0 waiting, 1 running, 1 held
```

```bash
node09 (530)% llq
<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>node09.492.1</td>
<td>peterm</td>
<td>6/6</td>
<td>01:37</td>
<td>R 50</td>
<td>express</td>
<td>node13</td>
</tr>
<tr>
<td>node09.492.0</td>
<td>peterm</td>
<td>6/6</td>
<td>01:37</td>
<td>C 50</td>
<td>express</td>
<td></td>
</tr>
</tbody>
</table>

1 job steps in queue, 0 waiting, 0 pending, 1 running, 0 held
```

```bash
node09 (531)% llq
<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>node09.492.0</td>
<td>peterm</td>
<td>6/6</td>
<td>01:37</td>
<td>C 50</td>
<td>express</td>
<td></td>
</tr>
<tr>
<td>node09.492.1</td>
<td>peterm</td>
<td>6/6</td>
<td>01:37</td>
<td>C 50</td>
<td>express</td>
<td></td>
</tr>
</tbody>
</table>

0 job steps in queue, 0 waiting, 0 pending, 0 running, 0 held
```

```bash
node09 (532)% cat test5.492.0.out
I am step 1, running on node13.stc.uk.ibm.com
```

```bash
node09 (533)% cat test5.492.1.out
I am step 2, running on node13.stc.uk.ibm.com
```

As a final example, we emphasize that the job steps can be very different, so that a complicated suite of programs could be run as a job consisting of multiple related job steps. Example 4-7 on page 33 shows how an initial serial step loads data. If this is successful, the main parallel step runs. If the parallel step succeeds, the results are post-processed. If either of the first two steps fails, the final step can clean up.
Example 4-7  Different step types

# @ step_name = step_1
# @ job_type = serial
# @ queue
# @ step_name = step_2
# @ job_type = parallel
# @ wall_clock_limit = 20
# @ node = 4
# @ network.MPI = csss,not_shared,US
# @ dependency = ( step_1 == 0 )
# @ queue
# @ step_name = step_3
# @ job_type = serial
# @ dependency = ( step_2 == 0 )
# @ queue
# @ step_name = step_4
# @ job_type = serial
# @ dependency = ( step_1 != 0 ) || ( step_2 != 0 )
# @ queue
case $LOADL_STEP_NAME in
step_1)
  
  # Serial job step to stage the data
  # from AFS to the local directory
  
  echo copying /afs/hur/u/peterm/data.quick.examp to mix3d.inp
  cp /afs/hur/u/peterm/data.quick.examp mix3d.inp && exit 0 || exit 1
  ;;
step_2)
  
  # We have the input data.
  # Run the parallel application
  
  /usr/bin/poe ./mix3d < mix3d.inp
  ;;
step_3)
  
  # The parallel application worked
  # Plot the results
  
  lpr -PColor mix3d.ps
  ;;
step_4)
  
  # Either the data access failed
  # or the parallel application failed.
  
  echo "Back to the drawing board :-("
  ;;
Example 4-7 produces the output shown in Example 4-8.

Example 4-8  Output from Example 4-7

```
node09 (542)% llsubmit test6.cmd
llsubmit: The job "node09.stc.uk.ibm.com.498" with 4 job steps has been submitted.
node09 (543)% llq

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>node09.498.0</td>
<td>peterm</td>
<td>6/6 02:32</td>
<td>R</td>
<td>50</td>
<td>express</td>
<td>node13</td>
</tr>
<tr>
<td>node09.498.1</td>
<td>peterm</td>
<td>6/6 02:32</td>
<td>NQ</td>
<td>50</td>
<td>express</td>
<td></td>
</tr>
<tr>
<td>node09.498.2</td>
<td>peterm</td>
<td>6/6 02:32</td>
<td>NQ</td>
<td>50</td>
<td>express</td>
<td></td>
</tr>
<tr>
<td>node09.498.3</td>
<td>peterm</td>
<td>6/6 02:32</td>
<td>NQ</td>
<td>50</td>
<td>express</td>
<td></td>
</tr>
</tbody>
</table>

4 job steps in queue, 0 waiting, 0 pending, 1 running, 3 held
```

```
node09 (544)% llq

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>node09.498.1</td>
<td>peterm</td>
<td>6/6 02:32</td>
<td>R</td>
<td>50</td>
<td>express</td>
<td>node16</td>
</tr>
<tr>
<td>node09.498.3</td>
<td>peterm</td>
<td>6/6 02:32</td>
<td>NQ</td>
<td>50</td>
<td>express</td>
<td></td>
</tr>
<tr>
<td>node09.498.2</td>
<td>peterm</td>
<td>6/6 02:32</td>
<td>NQ</td>
<td>50</td>
<td>express</td>
<td></td>
</tr>
<tr>
<td>node09.498.0</td>
<td>peterm</td>
<td>6/6 02:32</td>
<td>C</td>
<td>50</td>
<td>express</td>
<td></td>
</tr>
</tbody>
</table>

3 job steps in queue, 0 waiting, 0 pending, 1 running, 2 held
```

```
node09 (545)% llq

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>node09.498.3</td>
<td>peterm</td>
<td>6/6 02:32</td>
<td>R</td>
<td>50</td>
<td>express</td>
<td>node15</td>
</tr>
<tr>
<td>node09.498.2</td>
<td>peterm</td>
<td>6/6 02:32</td>
<td>NR</td>
<td>50</td>
<td>express</td>
<td></td>
</tr>
<tr>
<td>node09.498.1</td>
<td>peterm</td>
<td>6/6 02:32</td>
<td>RM</td>
<td>50</td>
<td>express</td>
<td></td>
</tr>
<tr>
<td>node09.498.0</td>
<td>peterm</td>
<td>6/6 02:32</td>
<td>C</td>
<td>50</td>
<td>express</td>
<td></td>
</tr>
</tbody>
</table>

1 job steps in queue, 0 waiting, 0 pending, 1 running, 0 held
```

```
node09 (546)% llq

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>node09.498.2</td>
<td>peterm</td>
<td>6/6 02:32</td>
<td>NR</td>
<td>50</td>
<td>express</td>
<td></td>
</tr>
<tr>
<td>node09.498.1</td>
<td>peterm</td>
<td>6/6 02:32</td>
<td>RM</td>
<td>50</td>
<td>express</td>
<td></td>
</tr>
<tr>
<td>node09.498.0</td>
<td>peterm</td>
<td>6/6 02:32</td>
<td>C</td>
<td>50</td>
<td>express</td>
<td></td>
</tr>
<tr>
<td>node09.498.3</td>
<td>peterm</td>
<td>6/6 02:32</td>
<td>C</td>
<td>50</td>
<td>express</td>
<td></td>
</tr>
</tbody>
</table>

0 job steps in queue, 0 waiting, 0 pending, 0 running, 0 held
```

```
node09 (547)% cat test6.498.0.out
copying /afs/hur/u/peterm/data.quick.examp to mix3d.inp
node09 (548)% cat test6.498.1.err
ATTENTION: 0031-408 4 tasks allocated by LoadLeveler, continuing...
```
4.2 Job filters

Job filters are a mechanism provided by LoadLeveler where a submitted job command file can be parsed for verification and then modified if necessary by a site program. The keyword below in the configuration file needs to enabled:

SUBMIT_FILTER = <program>

The named filter program is executed and the job file passed to it as standard input. The filter then returns its output via standard output, which is sent to the \texttt{llsubmit} command.

The Perl script in Example 4-9 is a very simple example of a job submission filter. The script examines the job command file to make sure that if the user does not specify a Pool in the requirements keyword then the line is modified to define a Pool entry of 1 and then written to standard output. A message is written to standard error to inform the user of the modification.

\textit{Example 4-9  Custom submit filter}

\begin{verbatim}
#!/usr/bin/perl

while (<STDIN>) {
    chomp($value = $_);
    if ( $value =~ /requirements/ ) {
        if ( $value !~ /pool/ ) {
            print STDERR "WARNING : No Pool value specified, defaulting to 1\n";
            print $value . " && (Pool == 1)\n";
        }
    } else {
        print "$value\n";
    }
}
\end{verbatim}
The filter script above is enabled by modifying the configuration file:

```bash
SUBMIT_FILTER = /u/loadl/filters/submit.pl
```

The following script is submitted with the job filter above in place.

```bash
#!/bin/ksh
# @ job_type = serial
# @ executable = /u/markr/CRUNCH/a.out
# @ requirements = (Arch == "R6000") && (OpSys == "AIX51")
# @ queue
```

Here we are informed that a pool has not been requested, so the filter has set a
default of pool 1 and subsequently submitted the job, as shown in the following
output.

```
$ llsubmit test.cmd
WARNING : No Pool value specified, defaulting to 1
llsubmit: Processed command file through Submit Filter: 
"/u/loadl/filters/submit.pl".
llsubmit: The job "sp4n01.msc.itso.ibm.com.1979" has been submitted.
```

Also available to the submit filter script are the following environment variables.

- **LOADL_ACTIVE**: The version of LoadLeveler
- **LOADL_STEP_COMMAND**: The job command file name
- **LOADL_STEP_ID**: The LoadLeveler job identifier
- **LOADL_STEP_OWNER**: The user ID of the job submitter

If using a submit filter is not desirable, a wrapper script which preprocesses the
job command file, before invoking `llsubmit`, could be used to provide similar
functionality.

### 4.3 Run-time environment variables

LoadLeveler exports a number of environment variables to both the job and also
the prolog and epilog programs, if they are used. This enables a job step to find
out information about itself. These variables all have names of the form
**LOADLXXXX**. Running a job containing the line

```
env|grep LOAD
```

produces the output

```
LOADL_STEP_COMMAND=test3.cmd
LOADL_STEP_ACCT=a00001
LOADL_STEP_GROUP=group1
LOADL_PID=17574
```
The use of LOADL_STEP_NAME has already been shown in the examples in Section 4.1, “Job steps” on page 28. The use of these variables in the prolog and epilog scripts is discussed later in Section 4.10, “Using prolog and epilog scripts” on page 53.

4.4 Managing the job environment

Users may choose to pass as much or as little of their interactive environment through to the job as they like, by using the # @ environment directive.

The job’s environment is generated by executing the user’s .profile (ksh) or .login (csh) file. In addition, the LOADLXXX environment variables described in Section 4.3, “Run-time environment variables” on page 36 are also set for the job environment, if the environment keyword is not specified. Any variables set by the user in the interactive session from which llssubmit is issued are not passed through to the job. For example, the following command copies the user’s environment variables into the job, with the exception of LEAVE_ME_OUT:

# @ environment COPY_ALL; !LEAVE_ME_OUT

This next command copies only the two specified variables into the job:

# @ environment $ADD_ME; $ME_TOO

This next example sets the specified environment variable in the directive itself:

# @ environment SET_ME_HERE=42

Section 9.3, “Mail notification” on page 155 shows how passing the DISPLAY variable explicitly into the job enables LoadLeveler to notify the submitting user when the job has completed by opening an X-window on the user’s terminal.
4.5 Using job command file variables

Job command file examples, which we have shown above, contain a number of special variables in the directive lines. The documented list of variables available is:

- $(host) The host name of the submitting machine. This is NOT the name of the machine which schedules the job. It is the name of the machine that ran the `llsubmit` command.
- $(hostname) The same as $(host).
- $(domain) The domain of the submitting machine.
- $(jobid) The number of the job, assigned by the scheduler.
- $(cluster) The same as $(jobid). Its use is deprecated.
- $(stepid) The number of the step, beginning with zero.
- $(process) The same as $(stepid). Its use is deprecated.

Other variables are available if the corresponding # @ variable appears in the job command file:

- $(executable) The executable to be run.
- $(base_executable) If $(executable) is set, then $(base_executable) is set to the name of the executable with the directory component removed.
- $(class) The job class.
- $(comment) A comment string. No blanks are allowed.
- $(job_name) The name of the job.
- $(step_name) The name of the step.
- $(schedd_host) Short name of schedd machine.
- $(schedd_hostname) Full name of the schedd machine.

The case of the variable name is ignored, but parentheses must be used as shown. $(Class) is valid, and is the same as $(class), but $class is not allowed.

These variables can only be used in the # @ directives and not in the body of the job command file. In a Korn shell script, $(comment) returns the value returned by the `comment` command, which probably does not exist.
In fact, although only the above variables are documented, any variable appearing in the job command file as `@ variable = value` can be used subsequently as `${variable}`, as long as value contains no spaces. It is possible, but probably undesirable, to create a file called `test.out.(Memory>=64)` by referring to the `${requirements}` variable.

Example 4-10 is a rather contrived example showing some of these features.

```
Example 4-10 Use of job command file variables
# @ job_name = test1
# @ step_name = step_1
# @ job_type = serial
# @ executable = /usr/bin/sleep
# @ argument = 60s
# @ initialdir = /u/peterm/LL_RESIDENCY/TESTS/TEST8
# @ input = /dev/null
# @ output = ${job_name}.${jobid}.${stepid}.${base_executable}.${argument}
# @ error = ${output}
# @ class = express
# @ notification = never
# @ comment = ${host}.${class}.${job_type}.${step_name}
```

Note that the incorrect argument to the `sleep` command forces some output on standard error, which is sent to the same file as standard output (Example 4-11). Also note that `${host}` has been evaluated as `node13`, even though the job is scheduled by `node09`.

```
Example 4-11 Sample use of job command file variables
node13 (518)% llsubmit test1.cmd
llsubmit: The job "node09.stc.uk.ibm.com.513" has been submitted.
node13 (519)% llq -l node09.513|fgrep Comment
  Comment: node13.express.serial.step_1
node13 (520)% cat test1.513.0.sleep.60s
Usage: sleep Seconds
```

### 4.6 Managing classes

Classes can be used as a means to control the allocation of resources to jobs, access to the resources by users or groups, and so on.

Classes are defined in the `LoadL_admin` file, optionally with some characteristics set in a default class stanza:

```
default: type = class
    wall_clock_limit = 2:00:00
short: type = class
```
class_comment = "10 minute jobs"
wall_clock_limit = 10:00,9:00
include_users = peterm
total_tasks = 32

long:
type = class
class_comment = "10 hour jobs"
wall_clock_limit = 10:00:00,9:30:00
include_users = markr

The full list of possible keywords that may be used in the class stanzas can be found in Appendix B, “Concise reference” on page 187. In summary, the following can be specified in the class stanzas:

- Users and groups to be included or excluded.
- Default consumable resources.
- Requirement to run with task 0 on a master node.
- Maximum numbers of jobs.
- Maximum numbers of nodes and tasks which can be requested.
- Priority and nice value.
- Process and job limits.

Having defined the classes and their attributes, the LoadL_config, or more usually the local configuration file, specifies which classes can run on which nodes:

MAX_STARTERS = 4
Class = short(4) long(4)

In this case, four jobs can be simultaneously active out of the list below, which is also an alternative way to specify the Class statement:

Class = { "short" "short" "short" "short" \
         "long" "long" "long" "long" }

The GANG Scheduler, introduced with LoadLeveler Version 3.1, has introduced more control over which classes can run together, with the two keywords START_CLASS and PREEMPT_CLASS. These are discussed in Section 7.6, “GANG” on page 106.

The class information can be displayed for the whole cluster, using the `llclass` command:

<table>
<thead>
<tr>
<th>Name</th>
<th>MaxJobCPU</th>
<th>MaxProcCPU</th>
<th>Free</th>
<th>Max</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d+hh:mm:ss</td>
<td>d+hh:mm:ss</td>
<td>Slots</td>
<td>Slots</td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>0+00:09:00</td>
<td>0+00:08:00</td>
<td>16</td>
<td>16</td>
<td>10 minute jobs</td>
</tr>
<tr>
<td>long</td>
<td>0+09:00:00</td>
<td>0+08:00:00</td>
<td>16</td>
<td>16</td>
<td>10 hour jobs</td>
</tr>
</tbody>
</table>

40 Workload Management with LoadLeveler
More detailed information may be obtained with the -l option:

$ llclass -l long
=============== Class long ===============
Name: long
  Priority: 0
  Include_Users: markr
  Admin:
  NQS_class: F
  NQS_submit:
  NQS_query:
  Max_processors: -1
  Maxjobs: -1
Resource_requirement: ConsumableCpus(1)
  Class_comment: 10 hour jobs
  Wall_clock_limit: 0+00:30:00, -1
  Job_cpu_limit: 0+09:00:00, -1
  Cpu_limit: 0+08:00:00, -1
  Data_limit: -1, -1
  Core_limit: -1, -1
  File_limit: -1, -1
  Stack_limit: -1, -1
  Rss_limit: -1, -1
  Nice: 0
  Free: 16
  Maximum: 16

The classes configured on a node can be determined with the following command:

$ llstatus -l node15|grep Classes
ConfiguredClasses   = short(4) long(4)
AvailableClasses    = short(3) long(4)
DrainingClasses     =
DrainedClasses      =

Note that llclass has no knowledge of the implications of the relationship between the number of classes configured and the value of MAX_STARTERS. For example, if a job is running in the short class, then only three short slots are available, as shown above. But by implication, only three long slots are available, because one of the four total slots is used; llclass would still show four long slots available.

Sometimes it is necessary to stop all work running on a node or scheduled by a node, or to stop any new job running so that the node can be taken out of service.
Jobs can be stopped immediately using the `llctl -h <node> flush` command. This terminates all jobs on the node, forcing a checkpoint for jobs with checkpointing enabled. Jobs are requeued if `# @ restart = no` is not in the job command file.

Example 4-12 shows the commands executed by the LoadLeveler administrator, with the corresponding output.

**Example 4-12  Administrator commands to flush a node**

```
$ llstatus sp4n01 sp4n05
Name                      Schedd  InQ Act Startd Run  LdAvg  Idle  Arch      OpSys
sp4n01.msc.itso.ibm.com   Avail     1   0   Run      1 0.00     0 R6000     AIX51
sp4n05.msc.itso.ibm.com   Avail     0   0   Idle     0 0.99  9999 R6000     AIX51
$ llctl -h sp4n01 flush
llctl: Sent flush command to host sp4n01.msc.itso.ibm.com.
$ llstatus sp4n01 sp4n05
Name                      Schedd  InQ Act Startd Run  LdAvg  Idle  Arch      OpSys
sp4n01.msc.itso.ibm.com   Avail     1   0   Flush    1 0.57     0 R6000     AIX51
sp4n05.msc.itso.ibm.com   Avail     0   0   Idle     0 0.99  9999 R6000     AIX51
$ llq
Id                       Owner      Submitted   ST  PRI  Class        Running On
------------------------ ---------- ----------- -- --- ------------ -----------
sp4n01.2029.0            peterm      6/7  18:10  R  50  express      sp4n05
1 job steps in queue, 0 waiting, 0 pending, 1 running, 0 held, 0 preempted
$ llstatus sp4n01 sp4n05
Name                      Schedd  InQ Act Startd Run  LdAvg  Idle  Arch      OpSys
sp4n01.msc.itso.ibm.com   Avail     1   0   Flush    0 0.49     0 R6000     AIX51
sp4n05.msc.itso.ibm.com   Avail     0   0   Run      1 0.14  9999 R6000     AIX51
```

Example 4-13 shows the commands executed by the user. The job sp4n01.2029 moves from Running (R) on sp4n01 to Checkpointing (CK), to Starting (ST) on sp4n05, and finally to Running (R) on sp4n05.

**Example 4-13  User's job being flushed**

```
sp4n01 (337)% llsubmit crunch_755.cmd
llsubmit: The job "sp4n01.msc.itso.ibm.com.2029" has been submitted. sp4n01 (338)% llq
Id                       Owner      Submitted   ST  PRI  Class        Running On
------------------------ ---------- ----------- -- --- ------------ -----------
sp4n01.2029.0            peterm      6/7  18:10  R  50  express      sp4n01
1 job steps in queue, 0 waiting, 0 pending, 1 running, 0 held, 0 preempted
sp4n01 (339)% llq
Id                       Owner      Submitted   ST  PRI  Class        Running On
------------------------ ---------- ----------- -- --- ------------ -----------
sp4n01.2029.0            peterm      6/7  18:10  CK 50  express      sp4n01
```

42 Workload Management with LoadLeveler
Chapter 4. Managing jobs

1 job steps in queue, 0 waiting, 0 pending, 1 running, 0 held, 0 preempted
sp4n01 (340)% llq

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4n01.2029.0</td>
<td>peterm</td>
<td>6/7</td>
<td>18:10</td>
<td>ST</td>
<td>50</td>
<td>express</td>
</tr>
</tbody>
</table>

1 job steps in queue, 0 waiting, 0 pending, 1 running, 0 held, 0 preempted
sp4n01 (341)% llq

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4n01.2029.0</td>
<td>peterm</td>
<td>6/7</td>
<td>18:10</td>
<td>R</td>
<td>50</td>
<td>express</td>
</tr>
</tbody>
</table>

If the administrator prefers to wait for a running job to complete, then it is possible to “drain” either all classes running on a node, or just those jobs belonging to a specific class, again using the `llctl` command:

llctl -h <node> drain startd [classlist|allclasses]

To drain means to complete all running jobs, but not to start any more jobs.

Note the following two possibilities:

If we issue drain startd, then the status of the startd daemon moves from Running or Busy, to Draining, to Drained when the running jobs have completed. But DrainingClasses and DrainedClasses in the output from `llstatus -l` are empty, and AvailableClasses still shows the classes as being available. This is shown in the output of Example 4-14. In other words, it is the startd daemon which is drained, rather than the individual classes.

**Example 4-14  Draining the startd daemon**

```
$ llstatus sp4n01 sp4n05
Name                      Schedd  InQ Act Startd Run LdAvg Idle Arch      OpSys
sp4n01.msc.itso.ibm.com   Avail     1   0 Run      1 0.46     1 R6000     AIX51
sp4n05.msc.itso.ibm.com   Avail     0   0 Idle     0 0.61  9999 R6000     AIX51
$ llctl -h sp4n01 drain startd
llctl: Sent drain command to host sp4n01.msc.itso.ibm.com.
$ llstatus sp4n01 sp4n05
Name                      Schedd  InQ Act Startd Run LdAvg Idle Arch      OpSys
sp4n01.msc.itso.ibm.com   Avail   1 0 Drning  1 0.54     2 R6000   AIX51
sp4n05.msc.itso.ibm.com   Avail  0 0 Idle  0 0.61  9999 R6000   AIX51
$ llq
Id             | Owner  | Submitted | ST | PRI | Class   | Running On |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4n01.2033.0</td>
<td>peterm</td>
<td>6/7</td>
<td>20:06</td>
<td>R</td>
<td>50</td>
<td>express</td>
</tr>
</tbody>
</table>

1 job steps in queue, 0 waiting, 0 pending, 1 running, 0 held, 0 preempted
$ llcancel sp4n01.2033.0
llcancel: Cancel command has been sent to the central manager.

$ llstatus sp4n01 sp4n05

<table>
<thead>
<tr>
<th>Name</th>
<th>Schedd</th>
<th>InQ</th>
<th>Act</th>
<th>Startd</th>
<th>Run</th>
<th>LdAvg</th>
<th>Idle</th>
<th>Arch</th>
<th>OpSys</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4n01.msc.itso.ibm.com</td>
<td>Avail</td>
<td>0</td>
<td>Drned</td>
<td>0</td>
<td>0.72</td>
<td></td>
<td></td>
<td>R6000</td>
<td>AIX51</td>
</tr>
<tr>
<td>sp4n05.msc.itso.ibm.com</td>
<td>Avail</td>
<td>0</td>
<td>Idle</td>
<td>0</td>
<td>0.61</td>
<td>9999</td>
<td></td>
<td>R6000</td>
<td>AIX51</td>
</tr>
</tbody>
</table>

$ llstatus -l sp4n01|fgrep Classes

- ConfiguredClasses = express(2)
- AvailableClasses = express(2)
- DrainingClasses =
- DrainedClasses =

If we issue `drain startd <class>` or `drain startd allclasses`, then the status of the startd daemon moves from Running or Busy to Idle when the running jobs have completed. It does not show Draining and Drained. However, DrainingClasses and DrainedClasses in the output from `llstatus -l` show the relevant classes. This is shown in the output of Example 4-15. In this case, it is the class or classes which are drained, rather than the startd daemon.

**Example 4-15  Draining a class**

```
$ llq
<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4n01.2034.0</td>
<td>peterm</td>
<td>6/7</td>
<td>20:10</td>
<td>R</td>
<td>50</td>
<td>express</td>
</tr>
</tbody>
</table>
```

1 job steps in queue, 0 waiting, 0 pending, 1 running, 0 held, 0 preempted

$ llstatus sp4n01 sp4n05

<table>
<thead>
<tr>
<th>Name</th>
<th>Schedd</th>
<th>InQ</th>
<th>Act</th>
<th>Startd</th>
<th>Run</th>
<th>LdAvg</th>
<th>Idle</th>
<th>Arch</th>
<th>OpSys</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4n01.msc.itso.ibm.com</td>
<td>Avail</td>
<td>0</td>
<td>Idle</td>
<td>0</td>
<td>0.07</td>
<td>9999</td>
<td></td>
<td>R6000</td>
<td>AIX51</td>
</tr>
</tbody>
</table>

$ llstatus -l sp4n05|fgrep Classes

- ConfiguredClasses = express(2)
- AvailableClasses =
- DrainingClasses = express
- DrainedClasses =

$ llctl -h sp4n05 drain startd express
llctl: Sent drain command to host sp4n05.msc.itso.ibm.com.

$ llstatus sp4n01 sp4n05

<table>
<thead>
<tr>
<th>Name</th>
<th>Schedd</th>
<th>InQ</th>
<th>Act</th>
<th>Startd</th>
<th>Run</th>
<th>LdAvg</th>
<th>Idle</th>
<th>Arch</th>
<th>OpSys</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4n01.msc.itso.ibm.com</td>
<td>Avail</td>
<td>0</td>
<td>Idle</td>
<td>0</td>
<td>0.03</td>
<td>9999</td>
<td></td>
<td>R6000</td>
<td>AIX51</td>
</tr>
<tr>
<td>sp4n05.msc.itso.ibm.com</td>
<td>Avail</td>
<td>0</td>
<td>Run</td>
<td>1</td>
<td>0.74</td>
<td>9999</td>
<td></td>
<td>R6000</td>
<td>AIX51</td>
</tr>
</tbody>
</table>

$ llstatus -l sp4n05|fgrep Classes

- ConfiguredClasses = express(2)
- AvailableClasses =
- DrainingClasses = express
- DrainedClasses =

$ llq

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.7 Managing limits

LoadLeveler job steps may have certain process limits associated with them. The following keywords may be used:

- `core_limit = hardlimit, softlimit`
- `cpu_limit = hardlimit, softlimit`
- `data_limit = hardlimit, softlimit`
- `file_limit = hardlimit, softlimit`
- `job_cpu_limit = hardlimit, softlimit`
- `rss_limit = hardlimit, softlimit`
- `stack_limit = hardlimit, softlimit`
- `wall_clock_limit = hardlimit, softlimit`

These keywords may either appear in a class stanza of the LoadL_admin file. Or they may be used in the following form in the user's job command file:

```
# @ xxx_limit = hardlimit, softlimit
```

The core, CPU, data, file, rss, and stack limits correspond to the standard UNIX process limits, as returned by the `ulimit` command in the Korn shell. Whereas `cpu_limit` applies to an individual process of a job step, `job_cpu_limit` limits the total CPU time used by all processes in a job step. `wall_clock_limit` is the elapsed time for the job step, used by the GANG and BACKFILL schedulers for efficiently scheduling parallel jobs.

The storage limits are expressed as integer[.fraction] [units]; integer and fraction are up to eight digits for LoadLeveler 2.2 and earlier, and units can be b, w, kb, kw, mb, mw, gb, or gw. The default unit is bytes, and the prefix is a power of 2 (for example, k=2^{10}), rather than a power of 10 (that is, k=10^3). In LoadLeveler
Version 3.1, to allow for 64-bit values, the further abbreviations tb, pb, and eb have been added, and the integer and fraction fields may be wide enough to allow 64-bit values to be specified. A word is defined as a 32-bit word of four bytes in all releases.

Time limits are expressed in the form:

[[hours:]minutes:]seconds[.fraction]

Fractions are rounded to the nearest second. Only 32-bit values may be specified for the time limits.

Also defined are the predefined keywords:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rlim_infinity</td>
<td>The largest positive number.</td>
</tr>
<tr>
<td>unlimited</td>
<td>Same as rlim_infinity.</td>
</tr>
<tr>
<td>copy</td>
<td>Copied from the user's environment at time of job submission. job_cpu_limit is then set to the same as cpu_limit. This cannot be used for wall_clock_limit.</td>
</tr>
</tbody>
</table>

Table 4-1 is reproduced from Loadleveler V2R2 for AIX: Using and Administering, SA22-7311, as it summarizes how the limits are set in various circumstances:

<table>
<thead>
<tr>
<th>If the hard limit...</th>
<th>Then the...</th>
</tr>
</thead>
<tbody>
<tr>
<td>is set in both the class stanza and the job command file.</td>
<td>smaller of the two limits is taken into consideration. If the smaller limit is the job limit, the job limit is then compared with the user limit set on the machine that runs the job. The smaller of these two values is used. If the limit used is the class limit, the class limit is used without being compared to the machine limit.</td>
</tr>
<tr>
<td>is not set in either the class stanza or the job command file.</td>
<td>user per process limit set on the machine that runs the job is used.</td>
</tr>
<tr>
<td>is set in the job command file and is less than its respective job soft limit.</td>
<td>job is not submitted.</td>
</tr>
<tr>
<td>is set in the class stanza and is less than its respective class stanza soft limit.</td>
<td>soft limit is adjusted downward to equal the hard limit.</td>
</tr>
</tbody>
</table>
When a job step reaches a hard limit, LoadLeveler sends a SIGKILL signal to a serial job, a SIGTERM signal to a POE parallel job, and pvm_halt is invoked for a PVM job. The job step is therefore terminated.

When a job reaches a soft limit, SIGXCPU (signal 24) is sent to the job. This is a trappable signal, and the user’s application has the option of handling this signal, so that a job may exit more gracefully.

As very simple example, a job command file containing the following lines:

```
# @ wall_clock_limit = 60, 30
# @ queue
trap "echo I have reached my soft limit, ignore it!" XCPU
sleep 90
```

This produces Cputime limit exceeded on standard error and I have reached my soft limit, ignore it! on standard output. A more sophisticated use of soft limits for a real application might be to trap SIGXCPU and write a restart file at the end of the current iteration before terminating. Enough time must be allowed between soft and hard limits to allow the job to finish cleanly.

### 4.8 Managing job queues

As described in Section 4.6, “Managing classes” on page 39, LoadLeveler has several mechanisms for limiting the jobs which may be executing on nodes, such as TASKS_PER_NODE, Class, START_CLASS, and so on. But by default, LoadLeveler places no limits on the number of jobs that a user or group may have simultaneously queued, being considered for dispatch by the scheduler, or executed.
There are several keywords that may appear in the LoadL_admin file to stop individual users monopolizing the queues and several dynamic expressions that can be used in the LoadL_config file to reduce the priority of queued jobs based on current utilization.

In the LoadL_admin file, the following keywords may appear:

- **max_jobs_scheduled** This may appear in a machine stanza and is applied to a machine running the schedd daemon. The value specified is the maximum number of jobs which this machine is able to have running (or start running) in the whole cluster at any one time. This value is applied, regardless of the availability of free resources within the cluster.

  If a cluster has fifteen machines, with two schedd machines, each of which sets max_jobs_scheduled = 10, then when one schedd machine is unavailable, the cluster can potentially have idle resources.

- **maxidle** This may appear in either a user or group stanza and specifies the maximum number of job steps that the user or group may have queued, which are being considered for dispatch.

  If a maxidle=10 applies to a user, either directly in the user or class stanza, or implicitly through a default stanza, and that user submits 100 jobs, then 10 will be considered for dispatch by the scheduler, and 90 will be in the NotQueued job state.

- **maxjobs** This may appear in either a user or group stanza and specifies the maximum number of job steps that the user or group may have Running at any one time. All other jobs will either be Idle or NotQueued.

- **maxqueued** This may appear in either a user or group stanza, and specifies the maximum number of job steps that the user or group may have either running, or being considered for dispatch.

- **max_node** This may appear in either a user or group stanza and specifies the maximum number of nodes that can be requested in a # @ node = n directive in a job command file. It does not affect requests for nodes using the superseded max_processors and min_processors directives.

- **max_processors** This may appear in either a user or group stanza and specifies the maximum number of processors that can be
requested using the max_processors and min_processors directives.

**total_tasks**

This may appear in either a user or group stanza and specifies the maximum number of tasks that can be requested in a # @ total_tasks directive.

Note the relationship between maxjobs, maxidle, and maxqueued. maxjobs is for Running or Starting jobs, maxidle is for Idle (queued) jobs, and maxqueued is Running plus Idle jobs. We emphasize that Idle, or queued, means being considered for dispatch by the scheduler – there is no limit on the number of jobs a user may submit.

As an example, suppose a user has maxjobs=4, maxidle=6, and maxqueued=8, and submits 20 jobs. Then there may be two jobs Running, six jobs Idle, and 12 jobs NotQueued, or four jobs Running, four Jobs Idle, and 12 jobs NotQueued. But NOT four jobs Running and six jobs Idle, because that would exceed the maxqueued=8 limit.

Within the LoadL_config file, the main mechanism for stopping users getting unfair resources is to factor in the number of job steps running and queued into the SYSPRIO expression. As well as the various queuing priorities discussed in Section 7.1, “Queuing priorities” on page 98, the SYSPRIO expression may also contain the following variables:

- **GroupQueuedJobs** The number of jobs either queued or running for the group
- **GroupRunningJobs** The number of jobs running for the group
- **GroupTotalJobs** The total number of jobs for the group
- **UserQueuedJobs** The number of jobs either queued or running for the user
- **UserRunningJobs** The number of jobs running for the user
- **UserTotalJobs** The total number of jobs for the user

As an example, the administrator could set:

```
SYSPRIO: 0 - 20*UserRunningJobs - 10*UserQueuedJobs - UserTotalJobs,
```

so that users with the most jobs running have their queuing priority reduced and are less likely to have a new job scheduled.

It is very important to note that if the SYSPRIO expression contains any of the queuing priorities, such as UserSysprio or QDate (the time of job submission), then the SYSPRIO for a job is fixed at job submission time and requires no updating. By default, SYSPRIO is NOT updated by the negotiator, and is only
evaluated at submission time. However, the above variables are dynamic, and will probably be changing as the jobs sit in the queue. The administrator must set \texttt{NEGOTIATOR\_RECALCULATE\_SYSPRIO\_INTERVAL} to force the Negotiator daemon to update SYSPRIO periodically.

### 4.9 Managing job execution

There are several mechanisms for influencing the behavior of a job that has started execution. LoadLeveler was originally developed to exploit spare cycles on workstations, which are typically idle for much of the time when not actively being used. LoadLeveler is still used in this way by a number of sites who have large numbers of serial jobs to process and a variety of machines; for example, desktops and dedicated servers, including SP parallel machines.

Before discussing how to customize the \texttt{LoadL\_config} file to achieve this goal, we need to explain the syntax used.

Variables may be defined by the administrator by using an expression of the form $\texttt{MINUTE} = 60$.

The \texttt{MINUTE} variable may then referenced as $(\texttt{MINUTE})$. The following example results in \texttt{SIXTY} being set to the value 61:

\begin{verbatim}
SIXTY=$(MINUTE)
MINUTE = 61.
\end{verbatim}

LoadLeveler has a number of built-in variables, which can be referred to by name, without needing to use the $(\texttt{VAR})$ syntax. Some, such as \texttt{Cpus}, the number of CPUs installed, are static. Others, such as \texttt{LoadAvg}, the Berkeley one-minute load average, are constantly updated by the LoadLeveler daemons.

There are three special LoadLeveler variables which are known as machine variables. They are:

- Arch
- Machine
- OpSys

They may be set using the format:

\begin{verbatim}
variable: $(value)
\end{verbatim}

The default \texttt{LoadL\_config} file contains the following statements:

\begin{verbatim}
MINUTE = 60
StateTimer = (CurrentTime - EnteredCurrentState)
\end{verbatim}
BackgroundLoad = 0.7
HighLoad = 1.5
StartIdleTime = 15*$\text{(MINUTE)}$
ContinueIdleTime = 5*$\text{(MINUTE)}$
MaxSuspendTime = 10*$\text{(MINUTE)}$
MaxVacateTime = 10*$\text{(MINUTE)}$

KeyboardBusy = KeyboardIdle < $\text{(POLLING\_FREQUENCY)}$
CPU_Idle = LoadAvg \leq $\text{(BackgroundLoad)}$
CPU_Busy = LoadAvg \geq $\text{(HighLoad)}$

# START: $(\text{CPU\_Idle}) \&\& \text{KeyboardIdle} > $(\text{StartIdleTime})$
# SUSPEND: $(\text{CPU\_Busy}) \mid $(\text{KeyboardBusy})$
# CONTINUE: $(\text{CPU\_Idle}) \&\& \text{KeyboardIdle} > $(\text{ContinueIdleTime})$
# VACATE: $(\text{StateTimer}) > $(\text{MaxSuspendTime})$
# KILL: $(\text{StateTimer}) > $(\text{MaxVacateTime})$

START: T
SUSPEND: F
CONTINUE: T
VACATE: F
KILL: F

The uncommented expressions, fixed at either TRUE or FALSE, mean that a job will be considered for dispatch on this node and left executing. In this context, the five keywords have the following meanings:

**START**
LoadLeveler considers dispatching a job to the machine if **START** is TRUE.

**SUSPEND**
LoadLeveler suspends the job if **SUSPEND** is TRUE.

**CONTINUE**
LoadLeveler resumes execution of a suspended job if **CONTINUE** is TRUE.

**VACATE**
LoadLeveler vacates a job if **VACATE** is TRUE. That is to say, a checkpoint is forced, and the job is removed from the node. If \# @ restart = no is not specified in the job command file, the job is requeued. If a checkpoint exists, the job restarts from the checkpoint; otherwise, it starts from the beginning again.

**KILL**
When **KILL** is TRUE, vacated jobs are removed from the node and requeued. No checkpoint is taken. This is used if a job that has been vacated is taking too long to write a checkpoint.
The commented alternate definitions will achieve the following:

- Start a job if the Load Average is less than 0.7 and the keyboard or mouse has not been touched for fifteen minutes.
- Suspend a job if the keyboard or mouse is touched or the Load Average goes above 1.5.
- The suspended job will resume execution if the keyboard and mouse are not touched for 5 minutes and the Load Average falls below 0.7.
- If a job stays suspended for more than ten minutes, it will be vacated.
- If that job takes more than ten minutes to vacate, it will be killed.

Other possibilities are to use user-defined macros and the current time to control job execution:

```
Evening = ((tm_hour >= 19) || (tm_hour < 7))
Daytime = ((tm_hour>= 7) && (tm_hour < 19))
START: $(Evening)
SUSPEND: $(Daytime)
CONTINUE: $(Evening)
```

The job class can also be factored in to this type of expression:

```
Evening = ((tm_hour >= 19) || (tm_hour < 7))
Daytime = ((tm_hour>= 7) && (tm_hour < 19))
Weekend = ((tm_wday == 6) || (tm_wday==0))
WeekDay = ((tm_wday >= 1) && (tm_wday<= 5))
PrimeClass = ($(Class) == “Prime”)  
EveningClass = ($(Class) == “Evening”) 
WeekendClass = ($(Class) == “Weekend”) 
```

```
START: $(PrimeClass) || ($(EveningClass) && $(Evening)) || $(WeekendClass) && $(Weekend) 
SUSPEND: ($(EveningClass) && $(Daytime)) || $(WeekendClass) && $(Weekday) 
CONTINUE: ($(EveningClass) && $(Evening)) || $(WeekendClass) && $(Weekend) 
```

These expressions can be different on different machines in the LoadLeveler cluster, in which case they can be put in the local configuration file.

The expressions are updated by the Startd daemon every POLLING_FREQUENCY seconds, which is five seconds by default.
4.10 Using prolog and epilog scripts

LoadLeveler allows the administrator to define prolog and epilog programs, which are automatically run before and after every job step. For example:

\[
\begin{align*}
\text{JOB_PROLOG} &= /u/loadl/jprolog \\
\text{JOB_EPILOG} &= /u/loadl/jepilog \\
\text{JOB_USER_PROLOG} &= /u/loadl/uprolog \\
\text{JOB_USER_EPILOG} &= /u/loadl/uepilog
\end{align*}
\]

These programs are spawned by the Starter daemon for each task, node, and step of each job. The JOB_PROLOG and JOB_EPILOG are run under the LoadLeveler user UID and GID. The JOB_USER_PROLOG and JOB_USER_EPILOG are run under the UID and GID of the submitting user.

The programs are run in the following order:

1. JOB_PROLOG
2. JOB_USER_PROLOG
3. The user's command file for the job step
4. JOB_USER_EPILOG
5. JOB_EPILOG

The JOB_PROLOG and JOB_EPILOG, running as the LoadLeveler administrator, run in an environment which only has the LoadLeveler environment run-time variables described in Section 4.3, "Run-time environment variables" on page 36 available. The JOB_USER_PROLOG and JOB_USER_EPILOG have all these environment variables available, and also the variables defined from /etc/environment.

In addition, the JOB_USER_PROLOG may set environment variables for the job by writing the lines `echo env ENV_VAR=VALUE` to standard output. Environment variables set in this way are available in the job script itself, and both the JOB_USER_EPILOG and JOB_EPILOG scripts. Environment variables set by the job script using the `# @` environment directive are also visible in the JOB_USER_EPILOG, but not in the JOB_EPILOG. Environment variables set within the body of the job script are not visible in either of the epilog.

It is necessary to be careful when producing any other output from the PROLOG and EPILOG programs. The JOB_USER_PROLOG and JOB_USER_EPILOG programs run as the LoadLeveler administrator, and do not have permission to write to the submitting user's directories. Also, the USER_PROLOG could write to the output file specified by the user in a `# @ output = <file>` directive. But when the job runs, it creates a new file, so the output of the USER_PROLOG would be lost.
The exit status from the earlier stages of the job are communicated to the EPILOG programs via the environment variables:

- LOADL_PROLOG_EXIT_CODE
- LOADL_USER_PROLOG_EXIT_CODE
- LOADL_JOB_STEP_EXIT_CODE

These variables are set to the exit codes of the corresponding earlier stages, multiplied by 256.

If the JOB_PROLOG exits with a non-zero return code, then only the JOB_EPILOG runs. If the JOB_USER_PROLOG exits with a non-zero return code, then both the JOB_USER_EPILOG and JOB_EPILOG are run. Both epilogs are also run if the job step exits with a non-zero return code.

*Loadleveler V2R2 for AIX: Using and Administering, SA22-7311* gives an extensive example of a prolog and epilog program. Simpler examples given in Example 4-16 and Example 4-17 show how the JOB_PROLOG may be set up to create temporary directories which last for the lifetime of the job and are then removed in the JOB_EPILOG. The names of the directories are made available to the user via the environment variables TMPDIR and WORKDIR. Note that the directory, local to each node, is created on all machines, but the global GPFS directory is only created and removed by the first node to appear in the LOADL_PROCESSOR_LIST.

**Example 4-16 Sample JOB_PROLOG**

```bash
#!/bin/ksh
#
# Sample LoadLeveler prologue program
#
# Create a directory locally on the node
#
echo env TMPDIR=/scratch/LoadL/$LOADL_STEP_ID
mkdir $TMPDIR > /dev/null 2>&1
#
# Create a directory globally in GPFS.
#
echo env WORKDIR=/GPFS/LoadL/$LOADL_STEP_ID
#
# We only need to do this on the first node.
#
FIRST=`echo ${LOADL_PROCESSOR_LIST} | cut -f1 -d" "`
FIRST_IP=`/usr/bin/host $FIRST | /bin/cut -f1 -d"," | /bin/cut -f3 -d" "`
VALUE=`/usr/bin/odmget -q "name=css0 and attribute=netaddr and value=$FIRST_IP" CuAt 2>/dev/null | /bin/wc -l`
if [ $VALUE -gt 0 ]; then
  mkdir $WORKDIR > /dev/null 2>&1
fi
```

```
Example 4-17 Sample JOB_EPILOG

#!/bin/ksh
#
# Sample LoadLeveler epilog program
#
# Only allow the first node in the processor
# list remove the GPFS directories.
#
FIRST=`echo ${LOADL_PROCESSOR_LIST} | cut -f1 -d" 
FIRST_IP=`/usr/bin/host $FIRST | /bin/cut -f1 -d"," | /bin/cut -f3 -d" 
VALUE=`/usr/bin/odmget -q "name=css0 and attribute=netaddr and value=$FIRST_IP"
CuAt 2>/dev/null | /bin/wc -l`
if [ $VALUE -gt 0 ]; then
  /bin/rm -rf $WORKDIR > /dev/null 2>&1
fi
#
# Remove TMPDIR - it's local everywhere
#
/bin/rm -rf $TMPDIR > /dev/null 2>&1
exit 0
Managing parallel jobs

The management of parallel jobs requires consideration for both system configuration and operation. In this chapter, we describe various scenarios for managing the parallel jobs. The following topics will be discussed:

- Task assignment
- Executing parallel jobs interactively using LoadLeveler
- Executing parallel jobs in Batch
- Managing the jobs in SMP nodes
- Use of consumable resources
- AIX WLM and LoadLeveler integration
5.1 Task assignment

Task assignment allows LoadLeveler to be instructed on how to assign tasks to nodes. The assignment is carried down the node list, and the list is ordered on the MACHPRIO expression value of the nodes. In this discussion, the order of MACHPRIO for the nodes is A, B, C, and D.

5.1.1 Node and total_tasks

Use of the `node` and `total_tasks` keywords will allocate the tasks evenly among the required nodes. If the `total_tasks` cannot be evenly divided amongst the nodes, then larger groups will be assigned to the first nodes in the list that can accommodate them. For example, if you specify the following keywords in the job command file, then the task assignment on the nodes could be similar to those shown in Table 5-1.

```plaintext
# @ node = 4
# @ total_tasks = 25
```

**Table 5-1  Task assignment - node and total_tasks**

<table>
<thead>
<tr>
<th>Node</th>
<th>Available initiators</th>
<th>Assigned tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>D</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

The task count is divided into three groups of seven and one group of four. Node A cannot be allocated a block of seven, as it has only four available initiators, so it is skipped. Node B is allocated seven, Node C is allocated seven, and Node D is also allocated seven. At this point, Node A can now be allocated the remaining four tasks.

5.1.2 Node and tasks_per_node

Blocks of the specified `tasks_per_node` value are assigned to each node requested. Here we are requesting four tasks on two nodes, for a total task count of eight:

```plaintext
# @ node = 2
# @ tasks_per_node = 4
```
5.1.3 Task geometry

You can allocate which tasks run together on the nodes, but cannot specify on which nodes these tasks are assigned.

```ksh
# @ task_geometry = { (0,7) (5,3) (6,2) (4,1) }
```

The task_geometry statement above is specifying that tasks 0 and 7 run on a node together, tasks 5 and 3 on a node together, and so on. In this situation, no other task keywords are necessary or allowed because the scheduler knows four nodes are requested for a total task count of eight. Attempts to specify any other task assignment keyword will cause the job to be rejected.

**Example 5-1  Job command file demonstrating task geometry**

```ksh
#!/bin/ksh
# @ job_type = parallel
# @ job_name = mytestgeometry
# @ executable = /usr/bin/poe
# @ arguments = sleep 300
# @ class = poe_l
# @ task_geometry = { (5,1,7,4) (3,6) (2) (0) }
# @ requirements = (Arch == "R6000") && (OpSys == "AIX51")
# @ queue
```

We submitted the job command file in Example 5-1. The output from `llq -l <jobstepid>` was post processed to strip out the node to task assignments for clarity.

Task Instance: sp6n01:5:css0(1,MPI,US,1M)
Task Instance: sp6n01:1:css0(2,MPI,US,1M)
Task Instance: sp6n01:7:css0(3,MPI,US,1M)
Task Instance: sp6n01:4:css0(4,MPI,US,1M)
Task Instance: sp6n03:3:css0(1,MPI,US,1M)
Task Instance: sp6n03:6:css0(2,MPI,US,1M)
Task Instance: sp6n11:2:css0(1,MPI,US,1M)
Task Instance: sp6n14:0:css0(1,MPI,US,1M)

With consumable resources enabled (see Section 5.5.2, “Node usage: to share or not to share” on page 79), the task assignments above match the machine resource status. All four processors of sp6n01 have been used; two processors from sp6n03 and one processor on both sp6n11 and sp6n14:

```
$ llstatus -R sp6n01 sp6n03 sp6n11 sp6n14
Machine Consumable Resource(Available, Total)
----------------- ------------------
sp6n01.msc.itso.ibm.com ConsumableCpus(0,4)
sp6n03.msc.itso.ibm.com ConsumableCpus(2,4)
sp6n11.msc.itso.ibm.com ConsumableCpus(0,1)
sp6n14.msc.itso.ibm.com ConsumableCpus(0,1)
```
5.1.4 Blocking

Blocking is the method of allocating tasks to nodes in specified block sizes. The allocation of these blocks occurs one at a time in order of machine priority. If there are tasks left over from the block assignment, then these are allocated to a single node. For example, if you specify the following keywords in the job command file, then the task assignment on the nodes could be similar to those shown in Table 5-2:

```
# @ blocking = 4
# @ total_tasks = 17
```

<table>
<thead>
<tr>
<th>Node Order</th>
<th>Available Initiators</th>
<th>Assigned Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5-2 Task assignment - blocking

With unlimited blocking, as many tasks as possible are allocated to each node. The order of tasks assignment in this case is evaluated first by the number of available initiators on the node and then by its MACHPRIO expression value. For example, if you specify the following keywords in the job command file, then the task assignment on the nodes could be similar to those shown in Table 5-3:

```
# @ blocking = unlimited
# @ total_tasks = 15
```

<table>
<thead>
<tr>
<th>Node Order</th>
<th>Available Initiators</th>
<th>Assigned Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5-3 Task assignment - unlimited blocking

The first node considered is C, because it has the highest number of available initiators; it receives 12 tasks. Node B is considered next and is allocated the remaining three tasks. In this case, nodes D and A do not receive any tasks, because no tasks remain to be allocated. The purpose of this approach is that you can allocate as many tasks to as few nodes as possible.
5.2 Interactive parallel

One common application of interactive parallel job classes is to provide a facility for running debugging sessions with tools such as pdbx and Totalview within a cluster.

In order to use the Parallel Operating Environment (POE) interactively across user space, you must define a default interactive class, in the LoadL_admin file:

```plaintext
default_interactive_class = inter
```

In the class stanza of the LoadL_admin file, we define the attributes of the interactive class to have softlimit of 25 minutes and a hard limit of 30 minutes:

```plaintext
inter:  type = class                     # class for interactive jobs
       wall_clock_limit = 30:00,25:00   # Needed for BACKFILL scheduler
       class_comment = "Interactive - 30 mins wall clock limit"
```

5.2.1 POE environment variables

POE supports the use of environment variables and their associated command line options. Appropriate POE values can be set via the system environment file /etc/profile. Upon logging in, the user will inherit the POE settings from the /etc/profile, at which point they are free to modify any variables for their own environment by redefining the personal login profile files:

```plaintext
export MP_EUIDEVICE=css0
export MP_EUILIB=us
```

In the above example, we set environment variables for the switch device as css0 and the protocol to use for the device as user space. The man page for POE details all the environment variables and their command line equivalents or, alternatively, list the options with the command `poe -help`.

5.2.2 Use of pools

Pools are a compatibility feature carried through from previous versions of LoadLeveler, when a separately installed resource manager was required.

Pools are useful for creating grouping of nodes to form subsets based on identical hardware or functionality. A node can belong to multiple pools, which allows you to define overlapping for flexibility. This function can also be simulated using the feature keyword in the local machine configuration file, as shown below:

```plaintext
Feature = { "Pool1" "Pool4" }
```
Pool lists are defined in a node's machine stanza located in the LoadL_admin file and hence has the advantage of being centrally administered from one file, in contrast to the feature keyword.

In Example 5-2, thin nodes sp6n14, sp6n13, sp6n12, sp6n10, and sp6n09 are assigned to pool 4, while nodes sp6n01 and sp6n04 and all the nodes in the cluster are assigned to pool 1.

**Example 5-2  Machine stanzas showing pool definitions**

```
sp6n14: type = machine
        adapter_stanzas = sp6sw14 sp6n14
        pool_list = 1 4
        alias = sp6sw14

sp6n13: type = machine
        adapter_stanzas = sp6sw13 sp6n13
        pool_list = 1 4
        alias = sp6sw13

sp6n12: type = machine
        adapter_stanzas = sp6sw12 sp6n12
        pool_list = 1 4
        alias = sp6sw12

sp6n11: type = machine
        adapter_stanzas = sp6sw11 sp6n11
        pool_list = 1 4
        alias = sp6sw11

sp6n10: type = machine
        adapter_stanzas = sp6sw10 sp6n10
        pool_list = 1 4
        alias = sp6sw10

sp6n09: type = machine
        adapter_stanzas = sp6sw09 sp6n09
        pool_list = 1 4
        alias = sp6sw09

sp6n03: type = machine
        adapter_stanzas = sp6sw03 sp6n03
        pool_list = 1 2
        alias = sp6sw03

sp6n01: type = machine
        central_manager = alt
        adapter_stanzas = sp6sw01 sp6n01
        pool_list = 1 2
        alias = sp6sw01
```

**Example**

Here a POE command is requesting the use of two nodes from pool 2 using user space on the switch:

```
export MP_EUILIB=us
```
export MP_EUIDEVICE=css0
$poe hostname -nodes 2 -tasks_per_node 1 -rmpool 2
sp4n01.msc.itso.ibm.com
sp4n05.msc.itso.ibm.com

5.2.3 Debugging

Occasionally, it is necessary to set the environment variable MP_INFOLEVEL or its equivalent command line option *ilevel*. This will generate different levels of debug information; by default, the value is 1. The levels of debug are shown in Table 5-4.

Table 5-4  MP_INFOLEVEL options

<table>
<thead>
<tr>
<th>Debug level</th>
<th>Description of output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Error</td>
</tr>
<tr>
<td>1</td>
<td>Warning and error</td>
</tr>
<tr>
<td>2</td>
<td>Information, warning, and error</td>
</tr>
<tr>
<td>3</td>
<td>Some diagnostics, information, warning, and error</td>
</tr>
<tr>
<td>4,5,6</td>
<td>High and low level diagnostics, information, warning, and error</td>
</tr>
</tbody>
</table>

Setting the debug level to 3 with the *ilevel* option demonstrates POE contacting LoadLeveler in order to submit the POE command via a job command file:

$poe hostname -nodes 2 -tasks_per_node 1 -rmpool 1 -ilevel 3
INFO: DEBUG_LEVEL changed from 0 to 1
D1<L1>: ./host.list file did not exist
D1<L1>: mp_euilib = us
D1<L1>: node allocation strategy = 1
INFO: 0031-364 Contacting LoadLeveler to set and query information for interactive job
D1<L1>: Job Command String:
  @ job_type = parallel
  @ environment = COPY_ALL
  @ requirements = (Pool == 1)
  @ node = 2
  @ tasks_per_node = 1
  @ node_usage = not_shared
  @ network.mpi = css0,not_shared,us
  @ class = inter_class
  @ queue

INFO: 0031-380 LoadLeveler step ID is sp4n01.msc.itso.ibm.com.1996.0
In order to run a POE command across all the nodes in an eight node system in pool 1, we issue the following POE command:

```
poe btat -nodes 8 -tasks_per_node 1 -rmpool 1 2>&1 &
```

Because it is an interactive session, the command is executed in the background. The job queue shows that the job is running with the interactive parallel class `inter`:

```
$llq
Id          Owner       Submitted   ST PRI Class Running On
-------------------- ---------- ----------- -- --- ------------ -----------
sp6en0.1747.0 markr 5/31 20:37 R 50 inter sp6n10
1 job steps in queue, 0 waiting, 0 pending, 1 running, 0 held, 0 preempted
```

A detailed output of the job step with the `llq -x -l sp6en0.1747.0` command shows the task to node assignments:

```
Node ----

Name : 
Requirements : (Pool == 1) && (Arch == "R6000") && (OpSys == "AIX51")
```
5.2.4 Use of IP

Interactive parallel jobs that use IP over the switch when no LoadLeveler pool is requested will not be managed by LoadLeveler. You need to specify the nodes to be used by POE. This can be done by specifying a host.list file. By default, POE will look for a file called host.list in the current directory. Attempting to use IP mode without informing POE which nodes to use will result in the following error message:

$ poe hostname -procs 2
ERROR: 0031-808 Hostfile or pool must be used to request nodes

We create a host.list, as in Example 5-3, with the following contents:

Example 5-3  host.list file
sp6n01
sp6n03
sp6n09
When attempting to use the host.list file, and the nodes listed in the host.list file are not specified in the user’s .rhosts file or in the system’s /etc/hosts.equiv file, then the job will fail with the error message shown in Example 5-4. However, note that LoadLeveler does not use the rsh program itself; it only uses the ruserok() system call to see if the user should be trusted to initiate processes remotely.

Example 5-4  IP job authorization failure

```bash
$poe hostname -procs 2 -ilevel 2 -euidevice css0 -euilib ip
ERROR: 0031-212  pmd: node sp6n01: user markr denied from access from host sp6n01
```

When using LoadLeveler pools, the IP job is managed by LoadLeveler.

```
$poe hostname -procs 2 -ilevel 2 -rmpool 2 -euidevice css0 -euilib ip
INFO: 0031-364  Contacting LoadLeveler to set and query information for interactive job
INFO: 0031-380  LoadLeveler step ID is sp6en0.msc.itso.ibm.com.69736.0
INFO: 0031-119  Host sp6n01.msc.itso.ibm.com allocated for task 0
INFO: 0031-119  Host sp6n03.msc.itso.ibm.com allocated for task 1
INFO: 0031-724  Executing program: <hostname>
INFO: 0031-724  Executing program: <hostname>
sp6n01
sp6n03
INFO: 0031-656  I/O file STDOUT closed by task 0
INFO: 0031-656  I/O file STDOUT closed by task 1
INFO: 0031-656  I/O file STDERR closed by task 0
INFO: 0031-656  I/O file STDERR closed by task 1
INFO: 0031-251  task 0 exited: rc=0
INFO: 0031-251  task 1 exited: rc=0
INFO: 0031-639  Exit status from pm_respond = 0
```

5.2.5 Using LoadLeveler command files

The environment variable MP_LLFILE or the `poe` command option `-llfile` sets the name of a LoadLeveler job command file to be used, as shown in Example 5-5 on page 67. For specific node allocation, you can use a LoadLeveler job command file together with a host.list file. If you do not specify a LoadLeveler pool, the nodes listed in the host.list file will be requested.
Example 5-5  LoadLeveler commands file

```plaintext
#@ job_type = parallel
#@ environment = COPY_ALL
#@ requirements = (Pool == 2)
#@ node = 2
#@ tasks_per_node = 1
#@ node_usage = not_shared
#@ network.mpi = css0,not_shared,ip
#@ class = inter
#@ queue
```

The command file is passed to the `poe` command, as shown in Example 5-6, with a debug level of 2.

Example 5-6  Using LoadLeveler command files with parallel interactive

```plaintext
$ poe date -llfile ll.file -ilevel 2
INFO: 0031-364 Contacting LoadLeveler to set and query information for interactive job
INFO: 0031-380 LoadLeveler step ID is sp6en0.msc.itso.ibm.com.69829.0
ATTENTION: 0031-408 2 tasks allocated by LoadLeveler, continuing...
INFO: 0031-119 Host sp6n03.msc.itso.ibm.com allocated for task 0
INFO: 0031-373 Using MPI for messaging API
INFO: 0031-375 Using ip for euilib
INFO: 0031-377 Using css0 for euidevice
INFO: 0031-119 Host sp6n01.msc.itso.ibm.com allocated for task 1
INFO: 0031-724 Executing program: <date>
INFO: 0031-724 Executing program: <date>
Tue Jun 5 18:49:02 EDT 2001
Tue Jun 5 18:49:02 EDT 2001
INFO: 0031-656 I/O file STDOUT closed by task 1
INFO: 0031-656 I/O file STDOUT closed by task 0
INFO: 0031-656 I/O file STDERR closed by task 1
INFO: 0031-656 I/O file STDERR closed by task 0
INFO: 0031-251 task 1 exited: rc=0
INFO: 0031-251 task 0 exited: rc=0
INFO: 0031-639 Exit status from pm_respond = 0
```
5.3 Batch parallel

In order to use a specified network protocol POE within a job command file, we are required to specify the network protocol settings. By default, IP over the specified network interface is used. This is specified by using the network.xxxx keyword in the job command file:

```
# @ network.protocol = network_type[, usage [, mode [, comm_level ] ] ]
```

The following protocols are supported:

- **MPI**
  Use the Message Passing Interface

- **LAPI**
  Use the Low-Level Application Programming Interface

- **PVM**
  Use the Parallel Virtual Machine; this must be used on and only in IP mode

**Note:** You can use `network.MPI` and `network.LAPI` in the same job step.

The terms from the above example can be defined as:

- **network_type**
  This can either be the adapter name, such as css0 or the network_type, if specified in the LoadL_admin file. When using adapter striping, the network_type should be css.

- **usage**
  States the adapter usage, shared or not_shared. If you wish to share the adapter with other tasks, then use shared, which is the default usage. Otherwise, specify not_shared.

- **mode**
  The communication system to be used by the protocol. The possible options are US (user space) or IP (Internet Protocol), which is the default. For every instance of US used by a task, css requires an adapter window. When MPI and LAPI protocols that require adapter windows are used by the same task, then two adapter windows per task are used. The number of adapter windows for use on SP_switch2 is 16 while for SP_switch it was four.

- **comm_level**
  This allows a user to predict the level of communication between tasks in order to assign adapter resources. This is only applicable to US. When requesting a high comm_level, it may impact the ability of the job to be scheduled, due to higher resource requirement that may not be available.
5.3.1 MPI - User space in shared mode

The job command file in Example 5-7 is submitted three times. In the command file, the usage mode of the switch adapter is not specified, because the default is shared.

Example 5-7 Job command file for user space in shared mode

```bash
#!/bin/ksh
# @ job_type = parallel
# @ job_name = 1n2p-100
# @ executable = /usr/bin/poe
# @ initial_dir = /u/markr/MIX3D
# @ arguments = mix3d
# @ input = Data/$(job_name)
# @ output = $(job_name).$(jobid).out
# @ error = $(job_name).$(jobid).err
# @ class = express
# @ node = 1
# @ tasks_per_node = 2
# @ network.MPI = css0,,US,low
# @ requirements = (Arch == "R6000") && (OpSys == "AIX51")
# @ queue
```

As shown below, two jobs are dispatched to node sp6n01; the third job remains queued, because there are no available starters remaining:

```
$ llq
Id                     Owner  Submitted ST PRI Class  Running On
----------------------- ---------- ----------- -- --- ------------ -----------
sp6en0.1802.0          markr  6/1  10:22  R  50  express     sp6n01
sp6en0.1803.0          markr  6/1  10:22  R  50  express     sp6n01
sp6en0.1804.0          markr  6/1  10:22  I  50  express

3 job steps in queue, 1 waiting, 0 pending, 2 running, 0 held, 0 preempted
```

5.3.2 MPI - User space in non-shared mode

In Example 5-8, we specify the usage mode of the adapter because we require the non-shared mode.

Example 5-8 Job command file for user space in non-shared mode

```bash
#!/bin/ksh
# @ job_type = parallel
# @ job_name = 2n2p-100
# @ executable = /usr/bin/poe
# @ initial_dir = /u/markr/MIX3D
# @ arguments = mix3d
# @ input = Data/$(job_name)
```
The job command file in Example 5-8 on page 69 is submitted twice. Although the node sp6n01 has four processors and the first job 1779 is only using two processors, the job has reserved the switch adapter for its own use. This prevents any other switch based MPI applications running on this node at the same time as job 1779, so job 1780 remains queued:

```
$ llq
```

However, we now submit a job that requests to use the ethernet adapter with IP. There are available starters on sp6n01, so job 1782 is dispatched to the node:

```
llsubmit: The job "sp6en0.msc.itso.ibm.com.1782" has been submitted.
$ llq
```

If you want to guarantee that no other job can use your node across any adapter, then the keyword below should be entered in your job command file:

```
# @ node_usage = not_shared
```

However, if you submit a job requesting either a dedicated adapters or nodes, and there are insufficient idle resources available, your job will not be dispatched until sufficient idle resources are available, as in the case of job 1780 and 1781.

### 5.3.3 MPI - IP mode over the switch

In order to use IP mode on the switch, the network.MPI keyword should be changed in the job command file, as shown in Example 5-9 and Example 5-10 on page 71.
Example 5-9  Job command for IP in non-shared mode over the switch

#!/bin/ksh
# @ job_type = parallel
# @ job_name = 2n2p-100
# @ executable = /usr/bin/poe
# @ initial_dir = /u/markr/MIX3D
# @ arguments = mix3d
# @ input = Data/$(job_name)
# @ output = ${job_name}.$(jobid)out
# @ error = ${job_name}.$(jobid).err
# @ class = express
# @ node = 2
# @ tasks_per_node = 2
# @ network.MPI = css0,not_shared,IP
# @ requirements = (Arch == "R6000") && (OpSys == "AIX51")
# @ queue

Example 5-10  Job command file for IP in shared mode over the switch

#!/bin/ksh
# @ job_type = parallel
# @ job_name = 2n2p-100
# @ executable = /usr/bin/poe
# @ initial_dir = /u/markr/MIX3D
# @ arguments = mix3d
# @ input = Data/$(job_name)
# @ output = ${job_name}.$(jobid)out
# @ error = ${job_name}.$(jobid).err
# @ class = express
# @ node = 2
# @ tasks_per_node = 2
# @ network.MPI = css0,,IP
# @ requirements = (Arch == "R6000") && (OpSys == "AIX51")
# @ queue

5.3.4  MPI - IP mode over ethernet

If the application you are running only uses a small amount of communication at
the beginning and the end, then using IP over Ethernet would be appropriate, as
shown in Example 5-11 on page 72 and Example 5-12 on page 72.
Example 5-11  Job command file for IP in non-shared mode over Ethernet

```sh
#!/bin/ksh
# @ job_type = parallel
# @ job_name = 2n2p-100
# @ executable = /usr/bin/poe
# @ initial_dir = /u/markr/MIX3D
# @ arguments = mix3d
# @ input = Data/$(job_name)
# @ output = $(job_name).$(jobid)out
# @ error = $(job_name).$(jobid).err
# @ class = express
# @ node = 2
# @ tasks_per_node = 2
# @ network.MPI = en0,not_shared,IP
# @ requirements = (Arch == "R6000") && (OpSys == "AIX51")
# @ queue
```

Example 5-12  Job command file for IP in shared mode over Ethernet

```sh
#!/bin/ksh
# @ job_type = parallel
# @ job_name = 2n2p-100
# @ executable = /usr/bin/poe
# @ arguments = mix3d
# @ input = Data/$(job_name)
# @ output = $(job_name).$(jobid)out
# @ error = $(job_name).$(jobid).err
# @ class = express
# @ node = 2
# @ tasks_per_node = 2
# @ network.MPI = en0,shared,IP
# @ requirements = (Arch == "R6000") && (OpSys == "AIX51")
# @ queue
```

5.4 LAPI

Low level Application Programming Interface (LAPI) is an IBM proprietary protocol designed to bypass the IP stack and interact with the SP switch hardware directly, resulting in performance gains. Examples on how to use LAPI are given in /usr/lpp/ssp/css/samples/lapi.
5.5 Managing SMP nodes

Scheduling jobs to a symmetrical multiprocessing machine has always been an issue. This is due to the fact that a job can use more resources than it requests, as in the case of an OpenMP job that uses one processor and requests multiple threads that will use multiple processors in the machine.

To alleviate this situation, LoadLeveler utilizes the facilities of the AIX Workload Manager (WLM) to enforce resource usage of CPUs and memory. WLM imposes resource restrictions on jobs that attempt to use more than their share of the available resources. LoadLeveler maintains a track of the available resources in the cluster, decreasing the amount when a job is scheduled and increasing the amount when a job leaves the system.

As in WLM, you can create 27 classes on each machine (the Central Manager will only schedule up to 27 jobs per machine at any time). Machines that have no resources defined will not have any limits imposed.

Under WLM, the class name for the job is the LoadLeveler starter process ID (PID). For jobs that span multiple nodes, the WLM class names will be different because the starter PIDs will be different. Since all processes forked from the starter PID are governed by the restrictions of that WLM class, the associated epilog and prolog process to the job will also have the same restrictions imposed if a resource contention exists.

When WLM is active under LoadLeveler, the WLM configuration /etc/wlm/current is symbolically linked to /etc/wlm/LoadL. It is recommended that you do not modify or interfere with the configuration, as this will defeat the object of WLM enforcing workload management for LoadLeveler.

5.5.1 Consumable resources

Consumable resources are simply the resources of a machine that are available to be used or consumed by a user job. This can be resources such as CPUs, memory, licenses or resources that are available across the cluster, such as floating software licenses. ConsumableCpus and ConsumableMemory are special consumable resources whose use can be enforced.

Scheduler configuration

You must specify which consumables are to be regarded as resources for scheduling in the LoadL_config file.

SCHEDULE_BY_RESOURCES= name ...
The name value may be machine resources, such as ConsumableCPUs, ConsumableMemory, or ConsumableVirtualMemory, defined in machine stanza (see machine configuration section below) or floating resources defined in the FLOATING_RESOURCES keyword:

\[
\text{FLOATING\_RESOURCES = name(count) ...}
\]

The above example details which consumable resources in the cluster are to be regarded as being shared between all the nodes in the cluster. Machine resources, such as CPU, memory, and virtual memory, are not valid entries for the FLOATING\_RESOURCES keyword.

To enforce the resource usage, the following keyword can be used:

\[
\text{ENFORCE\_RESOURCE\_USAGE = ConsumableCpus ConsumableMemory | deactivate}
\]

This keyword specifies that AIX WLM should be used to enforce CPU and/or real memory resources. The variables ConsumableCpus and/or ConsumableMemory must be specified in the SCHEDULE\_BY\_RESOURCES keyword in order for it to be enforced. If deactivate is specified, LoadLeveler will deactivate AIX WLM on all of the nodes in the LoadLeveler cluster.

\[
\text{ENFORCE\_RESOURCE\_SUBMISSION = true | false}
\]

When true, this keyword specifies that LoadLeveler should check all jobs at submission time for the resources keyword. The job’s resources keyword needs to have the same resources specified as the ENFORCE\_RESOURCE\_USAGE keyword in order to be submitted. If the value specified is false, no checking will be done, and jobs submitted without the resources keyword will not have resources enforced, and may run on machines with other jobs whose resources are enforced.

The underlying interaction with WLM can be logged by adding the flag D\_FULLDEBUG to the STARTD\_DEBUG option. WLM entries within the StartLog have a WLM() label as shown below:

\[
06/09 11:44:24  TI-18 WLM(createWLMClass): config = , class name = 52222 class description = sp4n01.msc.itso.ibm.com.2161.0
\]

**Machine configuration**

In order for the scheduler to allocate resources from a machine, you must give the machine’s resources in the LoadL\_admin file in the machine stanza. If you have a system with a large number of similar nodes, then use the default machine stanza to set a default resources value:

\[
\begin{align*}
\text{default:} & \quad \text{type = machine} \\
& \quad \text{resources = ConsumableCpus(1)} \\
\text{machine1:} & \quad \text{type = machine}
\end{align*}
\]
resources = ConsumableCpus(4)

**Requesting consumables**

When ENFORCE_RESOUREC_SUBMISSION is set to TRUE in the LoadLeveler configuration file, jobs must specify the resources each task is expected to consume. The consumable resources required by the job are specified in the job command file using the resources keyword. The job's resources keyword must contain at least the same resources as specified in the configuration file's ENFORCERESOURCE_USAGE keyword:

```text
# @ resources = ConsumableCPUs(num) ConsumableMemory(num units)
```

The ConsumableMemory and ConsumableVirtualMemory keywords allow for the use of a number and its associated unit. The units are identical to those used LoadLeveler data limits in Section 4.7, “Managing limits” on page 45. If no units are specified, then units are assumed to be in megabytes (the value is rounded up to the nearest integer). The value for ConsumableMemory should be greater than zero, while the ConsumableVirtualMemory value must be greater than zero as well as greater or equal to the image_size.

**WLM resource reporting**

The `wlmstat` command will report the amount of resources being used by each class. The following information is returned:

- The current CPU percentage consumed by the WLM class, per node.
- The total CPU time consumed by the WLM class, per node.
- The current physical memory percentage consumed by the WLM class, per node.
- The memory high water level with the maximum number of resident pages used by the WLM class since it was created.

**Scenario 1 - ConsumableCPUs**

Consider the following scheduler configuration that specifies that jobs have a resource keyword with a consumablecpus value:

```text
SCHEDULE_BY_RESOURCES = ConsumableCpus
ENFORCE_RESOURCE_SUBMISSION = TRUE
ENFORCE_RESOURCE_USAGE = ConsumableCpus
```

With WLM in operation, we submit an OpenMP job. Job A requests four CPUs and is instructed to run with 16 tasks. This is described in Example 5-13 on page 75.

**Example 5-13**  Job command file entries for OpenMP job

```text
#!/bin/ksh
# @ job_type = serial
```
At this point, there is no contention for resources on the node, so the threads use all 16 processors. The situation alters when we submit the same job again. Now we observe a steady decrease in the %CPU used by Job A and increase for Job B. Soon the jobs use an average of 50 percent CPU each. Job C now begins to run on the node that wishes to consume eight CPUs with 16 tasks. The average percentage CPU usage for Jobs A and B is seen to decrease to 25 and Job C averages 50 percent CPU. This trend is continued until Job B completes. At this point, WLM allocates the remaining unused CPU from Job B to Jobs A and C, according to their share value. At this point Jobs A and C now have an increased share and their average %CPU usage reflects this. Once Job A completes, the remaining job is allocated the remaining %CPU and increases to 100 percent, due to the lack of contention.

Table 5-5  WLM CPU allocation

<table>
<thead>
<tr>
<th>JOB</th>
<th>%CPU</th>
<th>25</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td>B</td>
<td>-</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td>C</td>
<td>-</td>
<td>50</td>
<td>66</td>
</tr>
</tbody>
</table>

Table 5-5 shows the predicted share allocation for the scenario. The CPU percentage was measured using the nmon application with a sample interval of 20 seconds:

/local/bin/nmon -f -t -r crunch-omp -s20 -c300

The nmon data file was sorted and post processed and the results can be seen in Figure 5-1. Although deviations from the ideal share value are observed, a good adherence to the ideal is maintained.
Chapter 5. Managing parallel jobs

Figure 5-1  Managing CPUs using LoadLeveler and WLM

The AIX `ps` command has been modified to allow the query and display of process information for WLM classes:

```
ps -ae -o class -o pid,user,pcpu,pmem,vsz,wchan,args
```

### Scenario 2 - ConsumableMemory

To demonstrate that WLM constrains the amount of physical memory a job may consume when a resource contention exists, we submit jobs A and B to a node whose total consumable memory is 300 MB. Job A requests 100 MB of memory, but actually allocates 150 MB. While there is no contention for memory, the 150 MB image remains resident in memory. WLM sets a desirable level for the job (class 25884) as 55:

```
CLASS tr i #pr MEM sha min smx hmx des rap urap npg
25884  0 1   3  33 100   0  58 100  55   25  383  38560
```

Job B requests the remaining 200 MB of memory. Now, there is contention for the available memory. The memory statistics reported by WLM show that Job A’s (class 25884) desirable level has been decreased to 18 percent based on the share allocation of the memory:

```
CLASS tr i #pr MEM sha min smx hmx des rap urap npg
25884  0 1   3  33 100   0  58 100  18  -29  659  38570
22848  0 1   6  7 200   0  58 100  36   67  168  10615
```
As Job B (class 22848) increases its memory consumption, Job A is seen to decrease towards its WLM desirable level:

CLASS | tr | i | #pr | MEM | sha | min | smx | hmx | des | rap | urap | npg
25884 | 0 | 1 | 3 | 21 | 100 | 0 | 58 | 100 | 18 | -7 | 546 | 24663
22848 | 0 | 1 | 6 | 43 | 200 | 0 | 58 | 100 | 36 | -8 | 551 | 50433

In order to achieve this level, the system begins to swap out the job until its resident size meets the consumable requirement, as shown in Figure 5-2.

Figure 5-2  Managing memory using LoadLeveler and WLM

Because Job A is spending a huge amount of time swapping out to virtual memory, the performance of Job A is severely affected. This situation will continue until physical memory is freed by another job exiting the system.

The desirable level for a class is calculated as follows:

\[
\text{requested memory size/consumable memory size} \times 100 = \text{share } \%
\]

\[
\text{desirable level (with no other jobs running)} \times \text{share } \%/100 = \text{new desirable level}
\]

The starting desirable level for job A, with no other jobs running, is 55. This will also be the starting desirable level any job arriving, such as Job B. As Job B enters the system, the desirable level is recalculated as a percentage of the share. At this point, the new desirable level is recalculated for Job A and Job B as follows:
5.5.2 Node usage: to share or not to share

In order to specify that the node should be shared with other job steps, the node_usage keyword should be used:

```bash
# @ node_usage = slice_not_shared | shared | not_shared
```

Consider the use of the keyword when one node with four processors is currently the only idle node and Job A is submitted with the options below:

```bash
# @ node_usage = <option>
# @ node = 1
# @ tasks_per_node = 2
# @ network.MPI = css0,,US,low
```

Job A is dispatched to the idle node. Shortly afterwards, Job B is submitted with the same node requirements but with the possible node_usage options below.

- **shared**  
  This is the default option and states that the job step can share nodes with other job steps. In this case, where Job A was submitted with node_usage set to shared, Job B will be despatched to the node.

- **not_shared**  
  This prevents other job steps from being scheduled to the node your job is currently using. Here, Job B will not be dispatched to the same node as Job A.

- **slice_not_shared**  
  The job step will not share nodes with other job steps while it is running in its own GANG time slice. When this option is used with LoadLeveler schedulers other than GANG, it will behave as not_shared.
Why not to share

The reason not to share is clear for uniprocessor nodes. However, there are reasons to use both shared and not_shared on multiprocessor nodes:

- **shared**
  - Not all processors or memory in use
  - Performance not an issue
- **not_shared**
  - Require all processors or all physical memory
  - Performance

5.5.3 Using shared memory

Tasks of a parallel job running within the same node will still use the switch adapter for communications. It is more efficient for the tasks to communicate via shared memory within the node. This can be enabled by setting the POE environment variable MP_SHARED_MEMORY=yes. Note that this is not the default. When the tasks are running not_shared on an SMP node, we recommend also setting MP_WAIT_MODE=poll for optimal performance.

We ran a simple ping-pong test job consisting of two tasks on a single NightHawk2 node with an SP_Switch2. Setting MP_SHARED_MEMORY=yes approximately doubled the bandwidth achieved, and halved the latency. For further details on POE, consult the manual Parallel Environment for AIX: Operations and Use, Vol 1, SA22-7425.
Checkpoint and restart

In this chapter, we introduce the concept of job checkpointing. We discuss the necessary LoadLeveler administration configuration and user job command file keywords in order to use this facility.

**Attention:** Before you consider using Checkpoint/Restart function, refer to the LoadL.README file in the /usr/lpp/LoadL/READMES for information on availability and support for this function.

Checkpointing should be considered a valuable addition to the toolkit for effective workload management. We discuss the following:

- Job type
- System initiated checkpoint
- User initiated checkpoints
- Configurations for checkpointing
- Environment variables for checkpointing
- Job command file keywords
- Checkpointing serial job
- Checkpointing parallel jobs
- Considerations for checkpointing jobs
6.1 Overview

LoadLeveler checkpointing is a method of taking a snapshot, at a particular moment in time, of the state of a job and writing this snapshot to a file. Jobs can be restarted from the last successful checkpoint file if it fails before normal completion due to power outage or hardware or software failures. The behavior of a job restarted from a checkpoint is essentially indistinguishable from the behavior that would have been seen had the checkpoint and restart never occurred.

AIX 5L now introduces support for checkpointing and restarting a process or process group within the AIX kernel itself. This has enabled LoadLeveler Version 3.1 to support a much more general and reliable checkpoint and restart capability for both serial and parallel jobs. Jobs consisting of shell scripts can be checkpointed and jobs with multithreaded executables can also be checkpointed.

6.2 Job types

In order for a job to be checkpointed, the AIX kernel requires that the environment variable CHECKPOINT=yes be set at the time of execution of the processes.

LoadLeveler understands two job types: job_type = serial and job_type = parallel. If the user specifies, in the job command file, that this job can be checkpointed, then LoadLeveler sets CHECKPOINT=yes and both of these job types may then be checkpointed.

For interactive parallel jobs that use the User Space protocol, POE creates and submit a LoadLeveler job, and this can be checkpointed by LoadLeveler.

Interactive parallel jobs that use IP over the switch and use a host file do not interact with LoadLeveler. These can still be checkpointed, but using the parallel environment commands poeckpt and poerestart.
6.3 Checkpointing types

There are two basic checkpointing types: system initiated and user initiated. System initiated checkpoints are initiated outside of the program being checkpointed, either at intervals, or by running an external command to initiate the checkpoint. User initiated checkpoints are checkpoints that are initiated from within a program by a system call. These may be either complete or partial, according to whether one or all tasks of the parallel job step initiate the system call.

6.3.1 System initiated checkpoints

System initiated checkpoints may be initiated by two methods:

1. By calling the `llckpt` command. This command can either be called by the job step owner, or by a LoadLeveler administrator. The most important flags are given in Table 6-1.

   Table 6-1  Checkpoint command line options

<table>
<thead>
<tr>
<th>llckpt command</th>
<th>Result after a successful checkpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>llckpt &lt;job step id&gt;</td>
<td>Continue. This is the default.</td>
</tr>
<tr>
<td>llckpt -u &lt;job step id&gt;</td>
<td>Place the job in a UserHold state.</td>
</tr>
<tr>
<td>llckpt -k &lt;job step id&gt;</td>
<td>Terminate the job.</td>
</tr>
</tbody>
</table>

2. At intervals, defined on a per machine basis, as described below in the following keywords:

   MIN_CKPT_INTERVAL = <seconds>
   MAX_CKPT_INTERVAL = <seconds>

   The checkpoint interval is defined either globally, or for each machine in their local config file, using the two parameters

   A checkpoint occurs when the MIN_CKPT_INTERVAL value is reached. The interval is increased by a factor of two and another checkpoint is taken. The checkpoint interval is repeatedly doubled until it reaches or exceeds MAX_CKPT_INTERVAL. The interval then remains at the same value of MAX_CKPT_INTERVAL.

   The default values for MIN_CKPT_INTERVAL and MAX_CKPT_INTERVAL are 900 seconds and 7200 seconds respectively. This means that checkpoints are taken after 900 seconds, 1800 seconds, 3600 seconds, 7200 seconds, and every 7200 seconds thereafter.
6.3.2 **Complete user initiated checkpoints**

This type of checkpoint is initiated when every task of a parallel program invokes the library call `mp_init_ckpt(MP_CUSER)`. The checkpoint is only initiated after all tasks have invoked `mp_init_ckpt`.

6.3.3 **Partial user initiated checkpoints**

One task initiates the checkpoint on behalf of all the others by invoking the library call `mp_init_ckpt(MP_PUSER)`. In this case, the task invoking `mp_init_ckpt` is suspended until after the checkpoint is completed. All other tasks are essentially treated the same as an asynchronous system initiated checkpoint.

6.4 **Configuration**

To exploit the checkpoint feature effectively, we need to make a few changes to the LoadLeveler configuration files. This section describes changes to the configuration files.

6.4.1 **Administration file settings**

The following keywords can appear in a class stanza in the `LoadL_admin` file:

- **ckpt_dir = <directory>**
  
  Specifies the directory to be used for checkpoint files for jobs that did not specify the directory name in the job command file, or via the environment variable `MP_CKPTDIR`.

- **ckpt_time_limit = hardlimit, softlimit**
  
  Specifies limits for the elapsed time allowed to complete a checkpoint. When the softlimit is reached, LoadLeveler attempts to abort the checkpoint but continue the job. if the hardlimit is exceeded before the checkpoint is aborted, then the job will be cancelled.

6.4.2 **Configuration file settings**

It might be necessary to delete old checkpoint files created by LoadLeveler. The keyword `ckpt_cleanup_program` allows a program to be run by the schedd daemon, at a specified interval `ckpt_cleanup_interval`. 
The following keywords can appear in the LoadL_config file:

- ckpt_cleanup_program
- ckpt_cleanup_interval

### 6.4.3 Job command file settings

The following keywords can appear in the user's job command file:

- checkpoint = interval | yes | no
- restart_from_ckpt = yes | no
- ckpt_file = <filename>
- ckpt_dir = <directory>
- restart_on_same_nodes = yes | no
- ckpt_time_limit = hardlimit, softlimit

### 6.4.4 Environment variables

The following environment variables may be set:

- MP_CKPTDIR=<directory>
- MP_CKPTFILE=<filename>

The directory for storing checkpoint files for jobs that did not specify a directory in the job command file or environment variable is specified with the following keyword:

```
CKPT_DIR = <directory>
```

This is placed in the class stanza of the LoadL_admin file. There can be a global definition in the default class stanza or a class can have its own CKPT_DIR defined.

### 6.4.5 Job command file keywords

To state when you want to checkpoint your program, the keyword is:

```
checkpoint = yes | interval | no
```

Jobs submitted with the checkpoint set at interval instructs LoadLeveler to perform a checkpoint at defined intervals.

If checkpoint=yes is set, then the job step can be checkpointed by a user initiated mechanism by calling the checkpoint library function in the application program or by the `llckpt` command.
When checkpoint=no is set, the job is not considered for checkpointing and any checkpoint function call or llckpt command will fail. This failure is shown below with the use of the llckpt command:

```
$ llckpt sp6en0.69852.0
llckpt: The llckpt command will wait for the results of the checkpoint operation on job step sp6en0.69852.0 before returning.
llckpt: Checkpoint FAILED for job step sp6en0.69852.0, with the following error: primary error code = 0, secondary error code = 00000000/00000000, error msg len = 151, error msg = LoadLeveler: 2512-840 Request to checkpoint job step sp6en0.msc.itso.ibm.com.69852.0 is ignored because checkpointing is not enabled for the job step
```

**Checkpoint file naming**

You are able to specify both the location and name of a checkpoint file. It is the responsibility of the user submitting the job to check if the location of the checkpointed files exist and has sufficient access permissions for the checkpoint operation to be successful. You can set the location and file using the following keywords:

- `ckpt_dir = <directory>`
- `ckpt_file = [<directory>]<file_name>`

When the naming assignment for the checkpoint directory and file name of both serial and batch parallel jobs are evaluated, the following rules apply:

- **Checkpoint directory**
  - `ckpt_file` value, if an absolute path is given
  - `ckpt_dir` value from the job command file
  - `ckpt_dir` value from the class stanza in the Load_admin file
  - The initial working directory

- **Checkpoint file name**
  - `ckpt_file` value from job command file
  - Default of `<jobname.><job_step_id>.ckpt`

The naming assignment for interactive parallel jobs differ because we need to deal with the environment variables passed through to POE and hence the `ckpt_file` and `ckpt_dir` from the job command file will be ignored:
• Checkpoint directory
  – MP_CKPTFILE environment variable value at submit time, if an absolute path is given
  – MP_CKPTDIR environment variable value at submit time
  – The initial working directory
• Checkpoint file name
  – MP_CKPTFILE environment variable value at submit time
  – Default of name.poe.ckpt.pid

**Restarting from a previous checkpoint**
Sometimes it is useful to restart a job from a previous checkpoint to check modifications to a calculation or because the job output might have been lost:

```restart_from_ckpt = yes | no```

If the checkpoint file is not accessible, the job will terminate and will be removed from the queue.

**Restarting with the same task geometry**
When you attempt to restart a job step from a checkpoint, LoadLeveler will use the checkpoint control file stored in the checkpoint directory to allocate the task geometry. If the job step is vacated and the checkpoint keyword is enabled, then the Scheduler and the Central Manager will generate a task geometry directive for the job step before it is requeued.

**Restarting on the same machines**
A job step that is restarting may have a requirement for the job to be run on its previous set of nodes. When a job is vacated as a result of a machine flush, the Scheduler and the Central Manager check the restart options. If the keyword below is detected, then a host list is formed and stored with the job step; the Central Manager then uses this list to schedule the job:

```restart_on_same_nodes = yes | no```

This option is only valid when a job is vacated; it is not valid when the job completes and is removed from the queue. If the job is resubmitted, ```restart_on_same_nodes=yes``` will be ignored. If the job is resubmitted with this option, it will be ignored. You have the ability to force this feature by either creating a host list or task geometry in the job command file on submission.
6.4.6 Environment variable effects

For processes that can be checkpointed, the process environment variable CHECKPOINT needs to be set to yes when the job is executed. This is not an issue for batch jobs, because this is taken care of by LoadLeveler. However, for interactive parallel jobs, the user must set this environment variable before the Parallel Operating Environment (POE) is requested to execute the process.

Table 6-2 details the effects of the environment variable CHECKPOINT and the job command file keywords with the level of checkpointing permitted.

Table 6-2 Effects of checkpoint values

<table>
<thead>
<tr>
<th>Environment Variable CHECKPOINT</th>
<th>YES</th>
<th>NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch</td>
<td>POE initiated</td>
<td>POE initiated</td>
</tr>
<tr>
<td></td>
<td>LL initiated</td>
<td>LL initiated</td>
</tr>
<tr>
<td></td>
<td>USER initiated</td>
<td>USER initiated</td>
</tr>
<tr>
<td>Interactive</td>
<td>POE initiated</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>LL initiated</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>USER initiated</td>
<td>N/A</td>
</tr>
<tr>
<td>checkpoint=no</td>
<td>Batch</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Interactive</td>
<td>POE initiated</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>USER initiated</td>
<td>N/A</td>
</tr>
</tbody>
</table>

6.5 Serial

In this section, we describe some scenarios for checkpointing serial jobs.

6.5.1 Scenario 1 - Checkpoint and restart on the same node

In Example 6-1, a user submits the serial job command file that instructs the job to take a checkpoint and is also instructed to restart on the same nodes from the specified checkpoint file if a problem occurs.

Example 6-1 Serial job command file with checkpoint and restart on same nodes

```bash
#!/bin/ksh
# @ job_type = serial
# @ executable = /u/markr/CRUNCH/a.out
# @ input = /dev/null
# @ output = crunch-cprst_2.$(cluster).$(process).out
# @ error = crunch-cprst_2.$(cluster).$(process).err
# @ initialdir = /u/markr/CRUNCH
```
# @ notify_user = markr@sp4en0.msc.itso.ibm.com
# @ class = serial_l
# @ notification = always
# @ checkpoint = yes
# @ ckpt_file = /u/markr/CRUNCH/crunch_cprst.$(cluster)
# @ restart_on_same_nodes = yes
# @ requirements = (Arch == "R6000") && (OpSys == "AIX51")
# @ queue

$ llsubmit crunch-cprst_2.jcl
llsubmit: The job "sp4n01.msc.itso.ibm.com.804" has been submitted.

$ llq
<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4n01.804.0</td>
<td>markr</td>
<td>5/24 14:39</td>
<td>R</td>
<td>50</td>
<td>serial_l</td>
<td>sp4n05</td>
</tr>
</tbody>
</table>

1 job steps in queue, 0 waiting, 0 pending, 1 running, 0 held, 0 preempted

The administrator discovers that he needs to isolate a node that is currently running the job sp4n01.804.0. The administrator issues a flush instruction to the LoadLeveler daemons on the node:

```
[root@sp4en0]:/> llctl -h sp4n05 flush
llctl: Sent flush command to host sp4n05.msc.itso.ibm.com.
```

The job immediately performs a checkpoint, as shown in the llq status below:

```
[root@sp4en0]:/> llq
<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4n01.804.0</td>
<td>markr</td>
<td>5/24 14:39</td>
<td>CK</td>
<td>50</td>
<td>serial_l</td>
<td>(alloc)</td>
</tr>
</tbody>
</table>

1 job steps in queue, 0 waiting, 0 pending, 1 running, 0 held, 0 preempted
```

The job then returns to the queue in the idle state because it is waiting to resume execution on the node sp4n05 that it was started on, which is now currently unavailable:

```
<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4n01.804.0</td>
<td>markr</td>
<td>5/24 14:39</td>
<td>I</td>
<td>50</td>
<td>serial_l</td>
<td>(alloc)</td>
</tr>
</tbody>
</table>

1 job steps in queue, 1 waiting, 0 pending, 0 running, 0 held, 0 preempted
```

Once the administrator has completed the necessary work, he resumes the daemons on node sp4n05 and the job is observed starting from the checkpoint:

```
<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
</table>

Chapter 6. Checkpoint and restart
Example 6-2 demonstrates the extended time separation in the output from the point where it was checkpointed and sat idle in the job queue until it the time when it was restarted on node sp4n05.

Example 6-2   program output after restart

<table>
<thead>
<tr>
<th>Starting iteration</th>
<th>Host</th>
<th>Time</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>sp4n05.msc.itso.ibm.com</td>
<td>14:39:40</td>
<td>05/24/01</td>
</tr>
<tr>
<td>35</td>
<td>sp4n05.msc.itso.ibm.com</td>
<td>14:39:42</td>
<td>05/24/01</td>
</tr>
<tr>
<td>36</td>
<td>sp4n05.msc.itso.ibm.com</td>
<td>14:39:43</td>
<td>05/24/01</td>
</tr>
<tr>
<td>37</td>
<td>sp4n05.msc.itso.ibm.com</td>
<td>14:46:30</td>
<td>05/24/01</td>
</tr>
<tr>
<td>38</td>
<td>sp4n05.msc.itso.ibm.com</td>
<td>14:46:31</td>
<td>05/24/01</td>
</tr>
<tr>
<td>39</td>
<td>sp4n05.msc.itso.ibm.com</td>
<td>14:46:32</td>
<td>05/24/01</td>
</tr>
</tbody>
</table>

6.5.2 Scenario 2 - Checkpoint and restart

In this example, we show a checkpoint and restart operation of a job by simulating a node down condition by flushing the LoadLeveler daemons, using `llctl` command. In this situation, a checkpoint operation is performed and restarted on a different node using the checkpoint file.

Example 6-3   Serial job command file with user specified checkpoint file

```bash
#!/bin/ksh
# @ job_type = serial
# @ executable = /u/markr/CRUNCH/a.out
# @ output = crunch-rst_1.out
# @ error = crunch-rst_1.err
# @ initialdir = /u/markr/CRUNCH
# @ class = serial_l
# @ notification = always
# @ notify_user = markr@sp4en0.msc.itso.ibm.com
# @ checkpoint = yes
# @ ckpt_file = /u/markr/crunch-1_ckpt
# @ requirements = (Arch == "R6000") && (OpSys == "AIX51")
# @ queue
```

The user submits the job command file in Example 6-3, and the job begins execution on node sp4n05:

```
$ llsubmit crunch-s_cpkt_1.jcl
llsubmit: The job "sp4n01.msc.itso.ibm.com.805" has been submitted.
```

The administrator then issues a flush daemon request to sp4n05. The job is checkpointed by the system and resumes execution on node sp4n05 from the point it was checkpointed previously on node sp4n05:
Chapter 6. Checkpoint and restart

The output from job sp5n01.805.0 below demonstrates the job restarting from a checkpoint on a different node. The time difference between iteration 36 and 37 was the time taken to initiate, complete and start from the checkpoint:

Starting iteration 34 on host sp4n05.msc.itso.ibm.com at 15:01:59 on 05/24/01
Starting iteration 35 on host sp4n05.msc.itso.ibm.com at 15:02:00 on 05/24/01
Starting iteration 36 on host sp4n05.msc.itso.ibm.com at 15:02:01 on 05/24/01
Starting iteration 37 on host sp4n01.msc.itso.ibm.com at 15:02:42 on 05/24/01
Starting iteration 38 on host sp4n01.msc.itso.ibm.com at 15:02:43 on 05/24/01
Starting iteration 39 on host sp4n01.msc.itso.ibm.com at 15:02:44 on 05/24/01

It should be noted that the short time taken for the checkpoint operations reflects the simplicity of the example job and the fact that checkpoint is performed to the local disk of the node.

6.5.3 Scenario 3 - Restarting from a previous checkpoint

Here we demonstrate the restarting of a job that terminated from its previous checkpoint file.

The user submits the serial job in Example 6-4. Several minutes into the process, the user takes a checkpoint. Several minutes later, after the checkpoint operation, the job fails when the node crashes.

Example 6-4 Serial job command file with user specified checkpoint file

```bash
#!/bin/ksh
# @ job_type = serial
# @ executable = /u/markr/CRUNCH/a.out
# @ output = crunch-1.out
# @ error = crunch-1.err
# @ initialdir = /u/markr/CRUNCH
# @ class = serial_l
# @ notification = always
# @ notify_user = markr@sp4en0.msc.itso.ibm.com
# @ checkpoint = yes
# @ ckpt_file = /u/markr/CRUNCH/crunch-1_ckpt
```
# @ requirements = (Arch == "R6000") && (OpSys == "AIX51")
# @ queue

Following the failure of the job, of which the user is informed by mail via the notification process, the user resubmits a job command file instructing LoadLeveler to restart from the checkpoint file, as shown in Example 6-5.

**Example 6-5  Serial job command file requesting a restart from a checkpoint**

```bash
#!/bin/ksh
# @ job_type = serial
# @ executable = /u/markr/CRUNCH/a.out
# @ output = crunch-1.out
# @ error = crunch-1.err
# @ initialdir = /u/markr/CRUNCH
# @ class = serial_l
# @ notification = always
# @ notify_user = markr@sp4en0.msc.itso.ibm.com
# @ checkpoint = yes
# @ restart_from_ckpt = yes
# @ ckpt_file = /u/markr/CRUNCH/crunch-1_ckpt
# @ requirements = (Arch == "R6000") && (OpSys == "AIX51")
# @ queue
```

### 6.6 Batch parallel

In this section, we describe the checkpoint function using the batch parallel programs.

#### 6.6.1 Scenario 1 - Checkpoint

In this scenario, we vacate a job on a node by flushing the LoadLeveler daemons. Example 6-6 shows a checkpoint operation is performed and then the job is restarted on a different node using the checkpoint file.

**Example 6-6  Parallel job command file with checkpoint enabled**

```bash
#!/bin/ksh
# @ job_type = parallel
# @ executable = /usr/bin/poe
# @ job_name = lnp2p-100
# @ arguments = /u/markr/MIX3D/mix3d
# @ input = MIX3D/Data/$(job_name)
# @ output = MIX3D/$(job_name).$(jobid).out
# @ error = MIX3D/$(job_name).$(jobid).err
# @ initialdir = /u/markr/MIX3D
```
The job starts on sp4n01, but then is flushed by the administrator. The system initiates a checkpoint, and the job then executes on sp4n05:

```bash
$ llsubmit mix3d-1n2p100-ckpt.jcl
llsubmit: The job "sp4n01.msc.itso.ibm.com.842" has been submitted.
$ llq
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4n01.842.0</td>
<td>markr</td>
<td>5/25 15:55</td>
<td>R</td>
<td>50</td>
<td>poe_l</td>
<td>sp4n01</td>
</tr>
</tbody>
</table>

1 job steps in queue, 0 waiting, 0 pending, 1 running, 0 held, 0 preempted

```bash
$ llq
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4n01.842.0</td>
<td>markr</td>
<td>5/25 15:55</td>
<td>CK</td>
<td>50</td>
<td>poe_l</td>
<td>sp4n01</td>
</tr>
</tbody>
</table>

1 job steps in queue, 0 waiting, 0 pending, 1 running, 0 held, 0 preempted

```bash
$ llq
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4n01.842.0</td>
<td>markr</td>
<td>5/25 15:55</td>
<td>R</td>
<td>50</td>
<td>poe_l</td>
<td>sp4n05</td>
</tr>
</tbody>
</table>

The job checkpoint information can be extracted from the output of the current job with the `llq -x -l` command. The output is described in Example 6-7.

### Example 6-7  Checkpoint information section from `llq -x -l`

<table>
<thead>
<tr>
<th>Checkpointable: yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ckpt Start Time:</td>
</tr>
<tr>
<td>Good Ckpt Time/Date:Fri May 25 15:56:18 EDT 2001</td>
</tr>
<tr>
<td>Ckpt Elapse Time: 56 seconds</td>
</tr>
<tr>
<td>Fail Ckpt Time/Date:</td>
</tr>
<tr>
<td>Ckpt Accum Time: 56 seconds</td>
</tr>
<tr>
<td>Checkpoint File: /GPFS4/markr/MIX3D/1n2p-100.842.ckpt</td>
</tr>
<tr>
<td>Restart From Ckpt: yes</td>
</tr>
<tr>
<td>Restart Same Nodes: no</td>
</tr>
<tr>
<td>Restart: yes</td>
</tr>
</tbody>
</table>

---

Chapter 6. Checkpoint and restart 93
6.6.2 Scenario 2 - Restarting from a checkpoint file

Restarting a parallel job from a checkpoint at submission time is not very different from that of a serial job (see Example 6-8).

Example 6-8  Parallel job command file requesting a restart from a checkpoint file

```
#!/bin/ksh
# @ job_type = parallel
# @ job_name = 2n1p-100
# @ executable = /usr/bin/poe
# @ arguments = /u/markr/MIX3D/mix3d
# @ input = Data/$(job_name)
# @ output = $(job_name).$(jobid).out
# @ error = $(job_name).$(jobid).err
# @ initial_dir = /u/markr/MIX3D
# @ class = poe_1
# @ notification = always
# @ notify_user = markr@sp4en0.msc.itso.ibm.com
# @ checkpoint = yes
# @ restart_from_ckpt = yes
# @ ckpt_file = /GPFS4/markr/MIX3D/$(job_name).$(jobid).ckpt
# @ node = 2
# @ tasks_per_node = 1
# @ network.MPI = css0,not_shared,US,low
# @ requirements = (Arch == "R6000") && (OpSys == "AIX51")
# @ queue
```

6.7 Considerations for checkpointing

Checkpointing is a very useful feature for scheduling another job when you are running a long job or you want to interrupt a job. However, it requires the resources of the currently running job. In the following list, we summarize here some important considerations for using the checkpoint feature:

- We recommend that you do not run jobs at the same time that you attempt to write to the same checkpoint file, as this will lead to file corruption.

- Deciding where jobs are to be checkpointed is crucial, both in terms of general availability and I/O bandwidth, which will affect the time taken to complete a checkpoint. We recommend the use of fast global storage, such as GPFS, to checkpoint files by nodes in a parallel job versus local storage, where you will need to move the checkpoint file to a different nodes or set of nodes in order to restart the job.

- When a job step is terminated and restarted, the accounting usage of the original process is restored.
When a job step is vacated, it is returned to the queue as idle and ready for rescheduling. If the job step has a valid checkpoint file, it will be restarted from that checkpoint and the wall clock limit will be adjusted accordingly.

If a job is cancelled while a checkpoint is in operation, the checkpoint operation will not complete.

Applications must be linked with the libpthreads.a library.

Only a one to one thread ratio is supported; any user specified thread ratios will be ignored.

Applications can only be restarted on another SP with compatible versions of AIX, PSSP, PE, and LoadLeveler.

Applications can only restart from checkpoints on machines with the same CPU and OS level (including fixes).

Any User Space applications that have been checkpointed with the SP_Switch (TB3, TB3MX, TB3MX2, or TB3PCI) cannot be restarted on the SP_Switch2; likewise, checkpoints taken on SP_Switch2 cannot be started on an SP_Switch.

When restarting an OpenMP job, the number of threads is preserved.

MPI applications must be linked with the threaded MPI library; otherwise, the checkpoint operation will fail.
Internal schedulers

LoadLeveler supports three internal schedulers. This chapter provides an overview of these schedulers and describes the circumstances when each of them may be used.

The LL_DEFAULT scheduler has been available in LoadLeveler since the product was first introduced.

The BACKFILL scheduler was introduced in LoadLeveler Version 2.1, to allow improved scheduling of parallel jobs.

The GANG scheduler has been newly introduced in LoadLeveler and offers a new paradigm for executing parallel jobs.

The use of external schedulers with LoadLeveler was introduced in LoadLeveler Version 1.3; this is discussed separately in Chapter 8, “Customized schedulers using API” on page 115.

Note that LoadLeveler’s management of queuing priorities is the same for each of the internal schedulers, so we discuss this first. We also discuss LoadLeveler’s node selection mechanisms at the outset, since it applies to all of the schedulers (with exceptions noted).
7.1 Queuing priorities

Jobs submitted to LoadLeveler have queuing priorities associated with them. These priorities may either be specified implicitly, using settings specified by the LoadLeveler administrator in the LoadL_admin file, or a priority may be specified by users in the job command file, enabling them to order their own jobs.

The following priorities are known to LoadLeveler and are available in the LoadL_config file for the administrator to use in ordering the queues:

- **ClassSysprio**: This is the priority defined by setting priority = n in a class stanza in the LoadL_admin file.
- **GroupSysPrio**: This is the priority defined by setting priority = n in a group stanza in the LoadL_admin file.
- **UserSysPrio**: This is the priority defined by setting priority = n in a user stanza in the LoadL_admin file.
- **UserPrio**: This is the priority defined by the user by setting user_priority = n in the job command file, or by using the `llprio` command.

These priorities may be used to construct a SYSPRIO expression in the LoadL_config file. Jobs are considered for dispatch in the order given by SYSPRIO. The default SYSPRIO expression is `SYSPRIO: 0 - (QDate)`, where QDate is the time of the job submission (specified as a number of seconds since LoadLeveler was started). This expression defines a first-in, first-out queuing scheme. Note that UserPrio should not be used in a SYSPRIO expression, as jobs are already ordered by UserPrio.

Note also that SYSPRIO only has an effect on the Central Manager node, so it makes no sense to set it in a local configuration file.

Other variables which may be used in the SYSPRIO expression are: UserRunningJobs, GroupRunningJobs, UserQueuedJobs, GroupQueuedJobs, UserTotalJobs, and GroupTotalJobs. Note that the priorities defined in the class, user and group stanzas of the LoadL_admin file do not change, and are fixed at job submission time. The variables immediately above are potentially constantly changing. If any of these variables are used in a SYSPRIO expression, the administrator should remember to set `NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL` to a positive value, to ensure that SYSPRIO is repeatedly re-evaluated.
Three commands are available to influence the queuing priorities of jobs waiting to execute:

**llprio**

This is used to change the UserPrio associated with a job and can therefore be issued by the owner of a job, as well as the LoadLeveler administrator. This command can be used to change the order of execution of queued jobs belonging to that user.

**llfavoruser**

Jobs of the favored user, sorted according to UserPrio, are given the highest priority, regardless of the SYSPRIO value. The current SYSPRIO value will be used to decide between users, if multiple users have been favored. The favored status is revoked with -u flag.

**llfavorjob**

The favored job(s) are given the highest priority. SYSPRIO is only used to decide between multiple favored jobs.

Note that the above three commands affect the order of queued jobs. They cannot alter the behavior of executing jobs and they are unconnected with the UNIX nice values. It should also be noted that the highest queuing priority does not guarantee that the job will run. If the required resources are not available, then it is possible that a lower priority, or unfavored job, whose resource requirements are met, will run before a high priority, or favored job, which requires resources which are currently unavailable.

The priorities of queued jobs may be examined using the `llq -l` command. The following values may be found in the output:

<table>
<thead>
<tr>
<th><strong>User Priority</strong></th>
<th>The value of UserPrio</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>user_sysprio</strong></td>
<td>The value of UserSysPrio</td>
</tr>
<tr>
<td><strong>class_sysprio</strong></td>
<td>The value of ClassSysPrio</td>
</tr>
<tr>
<td><strong>group_sysprio</strong></td>
<td>The value of GroupSysPrio</td>
</tr>
<tr>
<td><strong>System Priority</strong></td>
<td>The value of the SYSPRIO expression</td>
</tr>
<tr>
<td><strong>q_sysprio</strong></td>
<td>The value of the adjusted system priority. This is the value of SYSPRIO, but modified according to the UserPrio setting, to ensure that jobs belonging to the same user queued in the same class are ordered as specified by the user. In other words, users can guarantee the order in which their jobs will be ordered in the queue by setting user_priority appropriately, regardless of the value of SYSPRIO, which LoadLeveler evaluates.</td>
</tr>
</tbody>
</table>
Table 7-1 summarizes the relationship between these priorities and how they are specified.

Table 7-1  LoadLeveler priorities

<table>
<thead>
<tr>
<th>Value in LoadL_config</th>
<th>How to specify</th>
<th>Row label in “llq -l”</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserPrio</td>
<td>“user_priority = n” in a job command file</td>
<td>User Priority:</td>
</tr>
<tr>
<td>UserSysPrio</td>
<td>“priority = n” in a user stanza in LoadL_admin</td>
<td>user_sysprio:</td>
</tr>
<tr>
<td>ClassSysPrio</td>
<td>“priority = n” in a class stanza in LoadL_admin</td>
<td>class_sysprio:</td>
</tr>
<tr>
<td>GroupSysPrio</td>
<td>“priority = n” in a group stanza in LoadL_admin</td>
<td>group_sysprio:</td>
</tr>
<tr>
<td>SYSPRIO</td>
<td>“SYSPRIO: &lt;expression&gt;” in LoadL_config</td>
<td>System Priority:</td>
</tr>
<tr>
<td>N/A</td>
<td>(Calculated)</td>
<td>q_sysprio:</td>
</tr>
</tbody>
</table>

To show these priorities in action, we first submit a long-running job to fill up the available class slot on an SP node. Using the default SYSPRIO expression, we then submit ten jobs at ten second intervals (to ensure that QDate is different). These jobs are identical, except that they are given a user_priority of 10, 20, etc., giving the highest priority to the most recently submitted job.

The output from the llq command is shown below:

```
sp4n01 (1209)% llq
Id  Owner       Submitted  ST PRI Class       Running On
------------------------ ---------- ----------- -- --- ------------ -----------
sp4n01.1070.0  peterm      5/28 15:52 R  50  serial_l     sp4n01
sp4n01.1071.0  peterm      5/28 15:52 I  10  serial_l
sp4n01.1072.0  peterm      5/28 15:53 I  20  serial_l
sp4n01.1073.0  peterm      5/28 15:53 I  30  serial_l
sp4n01.1074.0  peterm      5/28 15:53 I  40  serial_l
sp4n01.1075.0  peterm      5/28 15:53 I  50  serial_l
sp4n01.1076.0  peterm      5/28 15:53 I  60  serial_l
sp4n01.1077.0  peterm      5/28 15:53 I  70  serial_l
sp4n01.1078.0  peterm      5/28 15:54 I  80  serial_l
sp4n01.1079.0  peterm      5/28 15:54 I  90  serial_l
sp4n01.1080.0  peterm      5/28 15:54 I 100  serial_l
```

11 job steps in queue, 10 waiting, 0 pending, 1 running, 0 held, 0 preempted
Job 1077 then had its priority increased by the user as follows:

```
sp4n01 (1272)% llprio +15 sp4n01.1077
llprio: Priority command has been sent to the central manager.
```

The output from the `llq` command after `llprio` is executed is shown below:

```
sp4n01 (1289)% llq
Id       Owner               Submitted    ST PRI Class       Running On
---------- ---------- ----------- -- --- ------------ -----------
sp4n01.1070.0 peterm 5/28 15:52 R  50  serial_l     sp4n01
sp4n01.1071.0 peterm 5/28 15:52 I  10  serial_l
sp4n01.1072.0 peterm 5/28 15:53 I  20  serial_l
sp4n01.1073.0 peterm 5/28 15:53 I  30  serial_l
sp4n01.1074.0 peterm 5/28 15:53 I  40  serial_l
sp4n01.1075.0 peterm 5/28 15:53 I  50  serial_l
sp4n01.1076.0 peterm 5/28 15:53 I  60  serial_l
sp4n01.1077.0 peterm 5/28 15:53 I  85  serial_l
sp4n01.1078.0 peterm 5/28 15:54 I  80  serial_l
sp4n01.1079.0 peterm 5/28 15:54 I  90  serial_l
sp4n01.1080.0 peterm 5/28 15:54 I 100 serial_l
```

11 job steps in queue, 10 waiting, 0 pending, 1 running, 0 held, 0 preempted

Job 1076 was then favored by the LoadLeveler administrator:

```
> llfavorjob sp4n01.1076.0
llfavorjob: Favorjob command has been sent to the central manager.
```

Note that the output from `llq -l` gives no indication that a job has been favored.

The long running job, 1070, was cancelled. The shorter jobs then ran, in the order shown in Table 7-2, which shows the values of the relevant priorities. This is extracted from the output of the `llq -l` command, before and after adjusting job 1077’s priority. Table 7-2 also shows the final order of execution, with the favored job executing first.

<table>
<thead>
<tr>
<th>Job #</th>
<th>UserPrio (before)</th>
<th>UserPrio (after)</th>
<th>SYSPRIO</th>
<th>q_sysprio (before)</th>
<th>q_sysprio (after)</th>
<th>Order of execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1070</td>
<td>50</td>
<td>50</td>
<td>-971</td>
<td>-1018</td>
<td>-1018</td>
<td>(executing)</td>
</tr>
<tr>
<td>1071</td>
<td>10</td>
<td>10</td>
<td>-978</td>
<td>-1069</td>
<td>-1069</td>
<td>10</td>
</tr>
<tr>
<td>1072</td>
<td>20</td>
<td>20</td>
<td>-988</td>
<td>-1059</td>
<td>-1059</td>
<td>9</td>
</tr>
<tr>
<td>1073</td>
<td>30</td>
<td>30</td>
<td>-998</td>
<td>-1049</td>
<td>-1049</td>
<td>8</td>
</tr>
<tr>
<td>1074</td>
<td>40</td>
<td>40</td>
<td>-1008</td>
<td>-1039</td>
<td>-1039</td>
<td>7</td>
</tr>
</tbody>
</table>

Chapter 7. Internal schedulers 101
### 7.2 Node selection

As with the queuing priority mechanism described in the previous section, the mechanism by which appropriate nodes are assigned to a job is independent of the underlying scheduler.

If a certain number of nodes are required for a particular job and there are more than that number of nodes available that meet all the requirements of that particular job, we can influence which node the job will run on by setting an expression for `MACHPRIO` in the `LoadL_config` file. As with `SYSPRIO`, `MACHPRIO` is only evaluated on the Central Manager node, so it makes no sense setting it in a local configuration file.

The default setting of `MACHPRIO` is `MACHPRIO: 0 - (LoadAvg)`, where `LoadAvg` evaluates to the Berkeley one-minute load average. So the lower the load on the node, the sooner it will be considered by LoadLeveler for running a job.

The full list of variables which may be included in a `MACHPRIO` expression is given in the *Loadleveler V2R2 for AIX: Using and Administering*, SA22-7311.

One useful variable is `CustomMetric`. There are several ways in which this can be used. The keyword `CUSTOM_METRIC` could be defined in the local configuration file, so that different nodes within the cluster can be assigned a different value. An alternative, and more general method, is to use the keyword `CUSTOM_METRIC_COMMAND` to specify a command to be run:

```
CUSTOM_METRIC_COMMAND = /u/loadl/CustomMetric
```

The command specified should produce no output, but should set a return code. This return code is then assigned by LoadLeveler to the `CustomMetric` variable.

<table>
<thead>
<tr>
<th>Job #</th>
<th>UserPrio (before)</th>
<th>UserPrio (after)</th>
<th>SYSPRIO</th>
<th>q_sysprio (before)</th>
<th>q_sysprio (after)</th>
<th>Order of execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1075</td>
<td>50</td>
<td>50</td>
<td>-1018</td>
<td>-1029</td>
<td>-1029</td>
<td>6</td>
</tr>
<tr>
<td>1076</td>
<td>60</td>
<td>60</td>
<td>-1029</td>
<td>-1008</td>
<td>-1008</td>
<td>1</td>
</tr>
<tr>
<td>1077</td>
<td>70</td>
<td>85</td>
<td>-1039</td>
<td>-998</td>
<td>-988</td>
<td>4</td>
</tr>
<tr>
<td>1078</td>
<td>80</td>
<td>80</td>
<td>-1049</td>
<td>-988</td>
<td>-998</td>
<td>5</td>
</tr>
<tr>
<td>1079</td>
<td>90</td>
<td>90</td>
<td>-1059</td>
<td>-978</td>
<td>-978</td>
<td>3</td>
</tr>
<tr>
<td>1080</td>
<td>100</td>
<td>100</td>
<td>-1069</td>
<td>-971</td>
<td>-971</td>
<td>2</td>
</tr>
</tbody>
</table>
7.3 Requirements and preferences

As described in Section 7.1, “Queuing priorities” on page 98, jobs waiting to be dispatched are ordered according to their various queuing priorities. But even when a job reached the top of the queue, there are several things that LoadLeveler needs to consider when deciding whether the job can run. First, there must be enough slots in the specified class available within the cluster to run the job. But jobs may have other requirements that need to be satisfied.

There are several keywords that may be specified in the job command file that specify requirements beyond simply a class and number of tasks. For both serial and parallel job, the user may specify additional requirements. The LoadLeveler manual gives several examples of the use of the requirements keyword. For non-homogeneous clusters, users might specify

```
# @ requirements = (Memory == 1024)
```
to run on nodes with a gigabyte of memory installed. If scratch files are written by a particular step of a multistep job to a local disk on a node, then it is possible to force subsequent steps to run on the same node by specifying

```
# @ requirements = (Machine == machine.step1)
```
requesting that this step runs on the same machine as was used for step1. For more general requirements, LoadLeveler allows machines to specify arbitrary features, by listing the features in the local configuration file:

```
# @ Features = { "Power3" "Nastran" }
```

A user may then specify

```
# @ requirements = (Feature == "Power3")
```
in the job command file.

For parallel jobs, the requirements for adapters, protocols, and so on, are specified using the network keyword. For example:

```
# @ network.MPI = css0,not_shared,US
```

Note that, prior to LoadLeveler Version 2.2, these communications requirements would have been specified as:

```
# @ requirements = (Adapter == "hps_us")
```

The preferences keyword follows the same syntax as requirements, and LoadLeveler will attempt to satisfy as many of the preferences as possible.

Note that, whatever scheduler is chosen, it is possible for a high priority job to sit at the top of the queue while lower priority jobs execute, because the requirements have not been met. LoadLeveler can indicate why a particular job is not running by using the `llq -s` command.

Note also that the use of preferences is NOT supported with the GANG scheduler.
7.4 Default scheduler

LoadLeveler's default scheduler is selected by specifying the following configuration in the Global Configuration file:

SCHEDULER_API = NO
SCHEDULER_TYPE = LL_DEFAULT

7.4.1 How it works

For serial jobs, LoadLeveler simply looks at the jobs at the top of the queue, and if there is a node available that satisfies any additional requirements, then the job will run. If there is more than one node available, then they are ordered according to the MACHPRIO expression. If preferences are specified, then the first node ordered by MACHPRIO which satisfies the preferences is selected.

Jobs may specify various limits, such as cpu_limit and wall_clock_limit, but these are normally not used in any scheduling decisions. The limits are only applied to the job once it is running on a node.

Note that an exception to this is when the administrator specifies a user stanza in the LoadL_admin file as follows:

```
peterm: type = user
  default_class = express short medium long
```

and

```
express: type = class
  cpu_limit = 05:00
short: type = class
  cpu_limit = 30:00
```

If user peterm submits a job without specifying a class, but specifies `# @ cpu_limit = 15:00`, then LoadLeveler will schedule the job in the short class, as this is the first class in the default_class list that satisfies the limits specified by the user.

For parallel jobs, the default scheduler attempts to accumulate sufficient resources to run the job. If a job requires N nodes, and they are available, then the job runs. If only M (<N) nodes are available, then these nodes are reserved by LoadLeveler. As jobs complete, more and more nodes are reserved for the job until enough nodes are available to run the job, at which time the job is dispatched.
Two keywords govern this reservation scheme:

- **NEGOTIATOR_PARALLEL_DEFER** specifies the time that elapses between attempts to reserve nodes for a job.
- **NEGOTIATOR_PARALLEL_HOLD** specifies the total amount of time that the job is allowed to reserve nodes before these nodes are dropped and returned to the pool of free nodes and the reservation process restarts.

There are two major disadvantages with this scheme. Firstly, as more and more nodes are reserved for a parallel job, the system as a whole becomes progressively less utilized. And secondly, parallel jobs often never manage to accumulate enough nodes within the **NEGOTIATOR_PARALLEL_HOLD** time limit in order to run.

### 7.5 Backfill

The BACKFILL scheduler provides more flexibility for scheduling parallel jobs and eliminates the disadvantages of the default scheduler. LoadLeveler’s BACKFILL scheduler is selected by specifying the following configuration in the Global Configuration file:

```
SCHEDULER_API = NO
SCHEDULER_TYPE = BACKFILL
```

One of the main differences between BACKFILL and the LL_DEFAULT scheduler from the user’s point of view is that jobs are obliged to specify a `wall_clock_limit`. Given a start time for a job and the maximum elapsed time, LoadLeveler can determine the corresponding latest end time for the job.

LoadLeveler examines the jobs that are currently executing and, based on the `wall_clock_limit` for the jobs, it determines the (latest) time at which they will end, freeing up the nodes. The scheduler then takes the highest priority queued job, which requires a certain number of nodes N. It is then able to look ahead and determine the time \( T_{\text{begin}} \) at which there will be enough nodes to run the highest priority job. But between now and \( T_{\text{begin}} \), there may be some free nodes, up to \( N-1 \), which can be used for running smaller jobs. If there are any free nodes, the scheduler looks for smaller jobs whose `wall_clock_limit` shows that they will finish before \( T_{\text{begin}} \), and these are dispatched. If there are no free nodes, then the scheduler waits for the next cycle.

As well as scheduling parallel jobs much more efficiently than the LL_DEFAULT scheduler, the BACKFILL scheduler also supports the scheduling of multiple tasks per node, and the scheduling of multiple User Space tasks per adapter. Neither of these functions are supported by the default scheduler.
7.6 GANG

LoadLeveler’s GANG scheduler is selected by specifying the following configuration in the Global Configuration file:

\[
\text{SCHEDULER\_API} = \text{NO} \\
\text{SCHEDULER} = \text{GANG}
\]

There are several other dependencies when the GANG scheduler is used. \text{PROCESS\_TRACKING} = \text{TRUE} must be set.

7.6.1 Introduction to GANG scheduling

Using either the LL\_DEFAULT or BACKFILL scheduler, a parallel job, once executed, continues to execute uninterrupted until completion (unless, of course, it is cancelled, or it fails for some reason). The GANG scheduler allows two or more jobs to be simultaneously active on a node and for execution to swap between the jobs at regular intervals. In the case of a parallel jobs, the switching between tasks is coordinated across the cluster.

As described in Section 3.7, “GANG scheduling” on page 21, the CSS software has been enhanced so that a process may relinquish a User Space window, but remain active within the system. Prior to this release, a process using User Space held on the window for the lifetime of the process. So even if a job could be suspended in some way, another job requiring a switch window could not be scheduled.

7.6.2 The GANG Matrix

Chapter 3, “New features” on page 17 introduced the concept of a GANG Matrix. In this section, we provide more details on this topic.

In the GANG Matrix, each column denotes one processor and each row represents one time slice. The number of columns is determined by the number of processors under the control of the GANG scheduler. The interval of the time slice is set by the LoadLeveler administrator to be the minimum length of execution for jobs such that the overhead of context switch does not significantly impact the overall system performance. The number of rows in the matrix, which is related to the multiprogramming level, will be dynamically adjusted by the scheduling system according to various factors:

- The requirements of the jobs in the system
- The real memory size of the processors
- The working set sizes of the applications running
- Limits and attributes set by system administration
Each entry in the matrix is a scheduling slot. Each scheduling slot is associated with a task of a running job or is empty. Jobs will occupy scheduling slots by rows such that their parallel tasks will context switch in and out of a set of processors simultaneously. The number of scheduling slots occupied by a job depends on several factors:

- The degree of declared parallelism
- The desired execution rate in comparing with other jobs

The organization of the GANG Matrix is illustrated in Figure 7-1. The figure shows a system of 12 nodes, where two nodes are four-way SMPs and the rest are uniprocessors. There are a total of 18 processors in the system, and, therefore, there are 18 columns in the GANG Matrix. Jobs A - J run in two logical space-sharing partitions. The first partition is comprised of machines 1 and 2 and runs jobs A and B. The multiprogramming level is 2 in this case. The second partition is comprised of machines 3 through 12 and runs jobs C through J. Its multiprogramming level is 4. In each logical partition, jobs are running in the time-sharing fashion on those processors. At the end of time slices, all the processors simultaneously but independently perform context switches.
The assignment of tasks to the slots of the matrix is also constrained by system characteristics. Each time slice of the GANG Matrix effectively corresponds to a virtual machine. This machine has resources that cannot be overcommitted by the tasks assigned to that slice. The most important resource is the number of user-space CSS windows available to user tasks. The number of user-space CSS windows used in a particular time slice cannot be greater than the total number available from CSS.

Another very important resource from a system perspective is virtual memory. The total amount of memory used by all the tasks (in all time slices) assigned to a node cannot be greater than the total virtual memory available in that node.

To support the coordinated context switches of tasks, as determined by the rows of the matrix, the local node-level scheduler on each machine recognizes the information sent from the GANG central manager (GCM). The information consists of the columns of the GANG Matrix corresponding to the processors in the machine, and a start time for the first slot in those columns.

The number of time slices in the matrix is dynamic. It is decided by the Scheduler for each instantiation of the matrix, according to:

- The number of jobs time-sharing each CPU
- The “execution_factor” parameter for each job

Since jobs are suspended and resumed as a result of the coordinated context switching, the elapsed time between job start and job completion is expected to increase, due to the time the job is suspended. The same expectation exists for jobs which have been preempted. This change in elapsed time is reflected in accounting data.

Each node that is running a task of a job needs to know the GANG Matrix. When the jobs is dispatched, the negotiator sends the GANG Matrix to the master daemon on the node running the startdd daemon for task 0. The master daemon extracts the relevant columns of the GANG Matrix and sends the rest of the GANG Matrix to one or more nodes. These nodes, in turn, extract the information relevant to themselves and then pass on the remaining information to one or more nodes. This proceeds in this hierarchical manner until all nodes for the job have the relevant columns of the GANG Matrix. The exact value of one or more nodes that define the communications hierarchy is defined by the value of the variable HIERARCHICAL_FANOUT in the LoadL_config file.

### 7.6.3 New GANG scheduler commands

In this section, we describe the new LoadLeveler commands for use with the GANG scheduler.
**llmatrix**

*llmatrix* may be used to list the GANG Matrix. This command is only available when the GANG scheduler is selected.

The command may specify a list of hosts:

```
llmatrix sp4n01 sp4n05
```

Or it may be called with no arguments, in which case the complete GANG Matrix information is displayed.

If the `-r` flag is used, then raw output is produced, where fields are separated by `!` (exclamation point). Otherwise, a formatted output is produced.

**llmodify**

*llmodify* may be used to modify the execution factor for a job step. Using `llmodify -x 2 sp4n01.168.0` will change the `execution_factor` to 2 for the specified job step.

### 7.6.4 Preemption using the GANG scheduler

The GANG scheduler and the enhanced CSS software have enabled jobs to be swapped over on an interval basis, as described in Section 7.6.2, “The GANG Matrix” on page 106. An alternative, and very important, application of the GANG scheduler has been to allow preemption, where a high priority job is allowed to swap out a lower priority job until the higher priority job has completed.

Preemption is required where a system may be full of potentially long-running jobs, yet a high priority job needs to run immediately. Such a scenario occurs in weather forecasting, where a machine may be used for research and development work at most times of the day, but a production forecast must be run according to a tight schedule – a late weather forecast is of no use!

A job may be preempted using two mechanisms, manually and automatically by class.

LoadLeveler Version 3.1 has introduced a new command `llpreempt`, which places the specified job step into the preempted state. The job remains in that state until the action is undone with `llpreempt -u`.

Jobs may be preempted by class using the PREEMPT_CLASS keyword. For example, the statement `PREEMPT_CLASS[A] = ALL {B C} ENOUGH{D E}` indicates to LoadLeveler that the job arriving in class A should preempt jobs in classes B, C, D, and E. ALL {B C} indicates that tasks belonging to jobs in
classes B and C are not allowed to execute in time-sharing mode with tasks of a class A job. ENOUGH(D E) means that tasks of the class D and E jobs may time-share with the tasks of the class A job. Only enough jobs in classes D and E will be preempted to allow the class A job to begin execution.

Note that preemption is only offered by the GANG scheduler.

### 7.6.5 Class starting rules using the GANG scheduler

With the LL_DEFAULT and BACKFILL schedulers, the number of jobs that may be started on a node is controlled by specifying MAX_STARTERS and Class in the LoadL_config file, or in the local configuration file:

- **MAX_STARTERS = 4**
- **Class = short(2) medium(2) long(2)**

This means that “any four jobs from the three classes, with a maximum of two jobs from each class.” Further control is available by using the ${Class} variable in the START expression in the configuration file, as discussed in Section 4.9, “Managing job execution” on page 50.

The GANG scheduler offers a more direct way of controlling classes and jobs, with the START_CLASS keyword. It is possible to say:

- **START_CLASS[A] = ( A == 0) && ( B < 2 ) && ( allclasses < 4 )**

This means to only start a class A job if there are no class A jobs already running, less than two class B jobs already running, and less than four jobs in any class already running. Note that allclasses is a LoadLeveler built-in name.

Note that for a job to start, both the START_CLASS and CLASS variables must evaluate to TRUE, and the rules specified in any PREEMPT_CLASS statement must also be satisfied.

### 7.6.6 GANG scheduling keywords

The GANG scheduler has introduced a number of new keywords. The following appear in the configuration file:

- **GANG_MATRIX_TIME_SLICE**
  (Default 60, minimum 60, maximum 3600) Specifies the amount of time that each row in the GANG Matrix will use to run jobs.

- **GANG_MATRIX_REORG_CYCLE**
  (Default 16) Specifies the number of “negotiation loops” that the scheduler waits to reorganize the GANG Matrix into disjointed subsets whose sizes are as close as possible to the ideal.
- **GANG_MATRIX_BROADCAST_CYCLE**
  (Default 300) Specifies the number of seconds that the scheduler waits to send the next GANG Matrix information to the stard daemons.

- **HIERARCHICAL_FANOUT**
  (Default 2) When GANG information is distributed, each node reads its local information, and passes on the remaining information to the number of children specified in this variable.

- **GANG_MATRIX_NODE_SUBSET_SIZE**
  (Default 512) Specifies the ideal number of nodes for subsets in the Gang Matrix.

- **START_CLASS**
  Defined above

- **PREEMPT_CLASS**
  Defined above

The following appear in the LoadL_admin file:

- **execution_factor**
  (Default 1, possible values: 1,2,3) This may appear in a class stanza. It specifies how much processing time jobs in this class will receive relative to other jobs time-sharing on the same nodes. For example, if job A has an execution_factor of 2 and job B has an execution_factor of 1, then LoadLeveler will allocate twice the number of rows in the GANG Matrix (and therefore twice the amount of processing time) to job A as to job B when A and B are executing on the same node.

- **max_smp_tasks**
  (Default = NCPUS) This may appear in a machine stanza. Specifies the maximum number of tasks that can be run concurrently on the machine.

- **max_total_tasks**
  (Default -1 [unlimited]) This may appear in a user, group or class stanza. Specifies the maximum total number of tasks that the scheduler will allocate at any given time to run the jobs of this user, group, or class.
7.6.7 Examples of the GANG scheduler

In this section, we describe a few example uses of GANG scheduling.

Context switching

In the LoadLeveler configuration file LoadL_config, we set the GANG time slice interval, stopped and then restarted the cluster:

GANG_MATRIX_TIME_SLICE=300

We submitted three jobs of the same class using 16 processors each. We drained sp4n05 to force the jobs to run on sp4n01:

$ llq

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4n01.951.0</td>
<td>markr</td>
<td>5/27 17:53</td>
<td>R</td>
<td>50</td>
<td>poe_1l</td>
<td>sp4n01</td>
</tr>
<tr>
<td>sp4n01.952.0</td>
<td>markr</td>
<td>5/27 17:53</td>
<td>R</td>
<td>50</td>
<td>poe_1l</td>
<td>sp4n01</td>
</tr>
<tr>
<td>sp4n01.953.0</td>
<td>markr</td>
<td>5/27 17:53</td>
<td>I</td>
<td>50</td>
<td>poe_1l</td>
<td>sp4n01</td>
</tr>
</tbody>
</table>

3 job steps in queue, 1 waiting, 0 pending, 2 running, 0 held, 0 preempted

The llmatrix command output shows that job 952 is currently using all 16 processors; the * denotes the active process:

$ llmatrix -r -s

sp4n01!Processor0!sp4n01.951.0*,sp4n01.952.0
sp4n01!Processor1!sp4n01.951.0*,sp4n01.952.0
sp4n01!Processor2!sp4n01.951.0*,sp4n01.952.0
sp4n01!Processor3!sp4n01.951.0*,sp4n01.952.0
sp4n01!Processor4!sp4n01.951.0*,sp4n01.952.0
sp4n01!Processor5!sp4n01.951.0*,sp4n01.952.0
sp4n01!Processor6!sp4n01.951.0*,sp4n01.952.0
sp4n01!Processor7!sp4n01.951.0*,sp4n01.952.0
sp4n01!Processor8!sp4n01.951.0*,sp4n01.952.0
sp4n01!Processor9!sp4n01.951.0*,sp4n01.952.0
sp4n01!Processor10!sp4n01.951.0*,sp4n01.952.0
sp4n01!Processor11!sp4n01.951.0*,sp4n01.952.0
sp4n01!Processor12!sp4n01.951.0*,sp4n01.952.0
sp4n01!Processor13!sp4n01.951.0*,sp4n01.952.0
sp4n01!Processor14!sp4n01.951.0*,sp4n01.952.0
sp4n01!Processor15!sp4n01.951.0*,sp4n01.952.0

On the next time slice, the output shows that job 951 is now active using all 16 processors:

$ llmatrix -r -s sp4n01

sp4n01!Processor0!sp4n01.951.0*,sp4n01.952.0
sp4n01!Processor1!sp4n01.951.0*,sp4n01.952.0
sp4n01!Processor2!sp4n01.951.0*,sp4n01.952.0
Preemption

We added the following preemption rules to the LoadLeveler configuration file LoadL_config, then stopped and restarted the cluster:

PREEMPT_CLASS[poe_l1] = ALL {allclasses}
PREEMPT_CLASS[poe_l2] = ALL {poe_14}

We submitted a two processor job of the class poe_l4 and this began running on sp4n01:

$ llq

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4n01.933.0</td>
<td>markr</td>
<td>5/27 15:21</td>
<td>R</td>
<td>50</td>
<td>poe_l4</td>
<td>sp4n01</td>
</tr>
</tbody>
</table>

1 job steps in queue, 0 waiting, 0 pending, 1 running, 0 held, 0 preempted

Following this action, we submitted a two processor job of the class poe_l2:

$ llsubmit mix3d-1n2p256.jcl
llsubmit: The job "sp4n01.msc.itso.ibm.com.938" has been submitted.

Job 938 can begin to start running and job 933 is shown as pending preemption:

$ llq

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4n01.938.0</td>
<td>markr</td>
<td>5/27 16:07</td>
<td>ST</td>
<td>50</td>
<td>poe_l2</td>
<td>sp4n01</td>
</tr>
<tr>
<td>sp4n01.933.0</td>
<td>markr</td>
<td>5/27 15:21</td>
<td>EP</td>
<td>50</td>
<td>poe_14</td>
<td>sp4n01</td>
</tr>
</tbody>
</table>

2 job steps in queue, 0 waiting, 2 pending, 0 running, 0 held, 0 preempted

Once Job 933 is preempted, job 938 now is seen as running:

$ llq

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Chapter 7. Internal schedulers  113
2 job steps in queue, 0 waiting, 0 pending, 1 running, 0 held, 1 preempted

Once job 938 completes and no other jobs preempt or prevent job 933 from running, it will return to the running state.

### 7.6.8 Comparison of the internal schedulers

As stated in Section 7.4, “Default scheduler” on page 104, the LL_DEFAULT scheduler is perfectly adequate for managing a varied serial workload. However, for effective scheduling parallel jobs, sites that have to schedule parallel jobs, but do not wish to use an external scheduler, should use either the BACKFILL or GANG scheduler.

In choosing the appropriate scheduler, the following points should be considered:

- Only the GANG scheduler allows coordinated context switching.
- Two of the new features of the GANG scheduler, namely automatic preemption based on classes using the PREEMPT_CLASS keyword, and the greater control of executing classes offered by the START_CLASS keyword, are likely to be of interest even where the coordinated context switching offered by GANG are not required. These two features are NOT available with the BACKFILL scheduler.
- The GANG scheduler, even when used with a one-row GANG Matrix, still uses the same internal algorithm for backfilling, and therefore should be just as efficient at maintaining high node utilization as the BACKFILL scheduler.
- If PREEMPT_CLASS and START_CLASS are not required, nor is coordinated context switching, then the BACKFILL scheduler has several advantages over GANG:
  - The GANG scheduler does not support Preferences.
  - The GANG scheduler still requires the GANG Matrix to be distributed, even though it is not used, resulting in a small overhead.
  - The BACKFILL scheduler can allow more than one task to be scheduled to a single CPU; the GANG scheduler only allows one task per CPU.
  - GANG scheduler requires synchronized clock (NTP) running on all the nodes.
Customized schedulers using API

Using LoadLeveler application programming interfaces (API), you can customize the use of LoadLeveler to suit your requirements. There are several external schedulers developed by customers for specialized use of LoadLeveler. In this chapter, we describe how to use the APIs and also focus on MAUI and SRS schedulers. The following topics will be discussed:

- Overview of LoadLeveler APIs
- Data access API
- External scheduler - MAUI
- External scheduler - SRS
8.1 LoadLeveler API

By using LoadLeveler Application programming interfaces (API), you can customize the use of LoadLeveler to suit your requirements. The customers can call these APIs inside their application programs to interact with the LoadLeveler environment and obtain specific data that may not be provided by the standard LoadLeveler commands. Others may be interested in getting the data in a particular format of their interest. By using the APIs, you can write your own scheduler, create new customized LoadLeveler commands, obtain a usage report in a format that may suit you, and so on. Table 8-1 provides the list of supported LoadLeveler APIs.

Table 8-1  LoadLeveler APIs

<table>
<thead>
<tr>
<th>LoadLeveler API</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data access API</td>
<td>To access the data from LoadLeveler objects</td>
</tr>
<tr>
<td>Accounting API</td>
<td>Account validation and retrieve data</td>
</tr>
<tr>
<td>Serial checkpoint API</td>
<td>User initiated checkpointing of serial jobs</td>
</tr>
<tr>
<td>Submit API</td>
<td>Submitting jobs</td>
</tr>
<tr>
<td>Workload management API</td>
<td>Provide tools to be used with external scheduler</td>
</tr>
<tr>
<td>Query API</td>
<td>Provides information about jobs and machines</td>
</tr>
</tbody>
</table>

For using APIs, you must include the header file llapi.h located in the /usr/lpp/LoadL/full/include directory. The APIs are not thread safe; they should not be linked by a threaded application.

8.1.1 Data access API

The data access APIs gives access to the LoadLeveler objects and allows you to retrieve specific data from the objects. Using this API, you can query the negotiator daemon for information about its current set of jobs and machines. The data access APIs consist of eight subroutines (see Table 8-2).

Table 8-2  Data access API subroutines

<table>
<thead>
<tr>
<th>Data access API subroutines</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ll_query</td>
<td>Initializes the query object and defines the type of query (jobs, machines or cluster) to perform.</td>
</tr>
</tbody>
</table>
Refer to Loadleveler V2R2 for AIX: Using and Administering, SA22-7311 for detailed description on each subroutine. This documentation also provides the example of using data access API. In this redbook, we provide some sample tools using the data access API, which are widely used in the field by customers and IBM LoadLeveler experts.

**Sample tools using data access API**

We tested two examples during this project, that are described in the following sections. These examples are designed to show how you can use the Data access APIs to get the information from the LoadLeveler objects. The source code and the makefile for these tools are provided in Appendix A, “Tools” on page 171.

**llbeans**

*llbeans* is a C code that queries the LL data API in order to extract job queue information from the system. The format of the call is:

```
llbeans <-r -w -v>
```

- **-r** prints running jobs
- **-w** prints waiting jobs
- **-v** prints a node list for any running job

The default flags for *llbeans* are -r and -w.

The *llq* output of a job submitted and running on the system is shown in Example 8-1.

**Example 8-1   Default llq output**

```
[shecar@fl8n13e]/u/home/shecar/JoeS]$ llq
Id            Owner      Submitted   ST PRI Class        Running On
------------------------ ---------- ----------- -- --- ------------ ---------
```

<table>
<thead>
<tr>
<th>Data access API subroutines</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ll_set_request</td>
<td>Filters the object you want to query.</td>
</tr>
<tr>
<td>ll_reset_request</td>
<td>Resets the request data to NULL.</td>
</tr>
<tr>
<td>ll_get_objs</td>
<td>Retrieves the list of objects from the LoadLeveler Daemons.</td>
</tr>
<tr>
<td>ll_get_data</td>
<td>Retrieves the specific data from the object.</td>
</tr>
<tr>
<td>ll_next_objs</td>
<td>Retrieves the next object in the list.</td>
</tr>
<tr>
<td>ll_free_objs</td>
<td>Frees all the objects you received.</td>
</tr>
<tr>
<td>ll_deallocate</td>
<td>To end the query.</td>
</tr>
</tbody>
</table>
This job requires four nodes and four tasks per node. The output of `llbeans` is illustrated in Example 8-2.

**Example 8-2 Sample llbeans output**

```
[shecar@f18n13e](/u/home/shecar/JoeS)$ llbeans -v
Job: f18n13s.277.0 wuzza 3 07/21/00-09:08:57 07/21/00-20:09:02 16 shecar No_Class
Nodes: f01n05s f01n06s f01n07s f01n10s
```

If you do not want to see the nodes, eliminate the `-v` flag, as shown in Example 8-3.

**Example 8-3 Description of llbean output**

```
[shecar@f18n13e](/u/home/shecar/JoeS)$ llbeans
job: f18n13s.277.0 wuzza 3 07/21/00-09:08:57 07/21/00-20:09:02 16 shecar No_Class
A           A         A               A             A   A  A
|           |         |               |             |   |      |
LL JobID       |         |               |             |      tasks   |      |
job_name  |         |               |             |            owner    |
job state   |         |               |             |                   job class
job queue time       |
job est. completion time
```

Example 8-4 shows `llq` with running and waiting jobs.

**Example 8-4 llq output for jobs**

```
shecar@f18n13e](/u/home/shecar/JoeS)$ llq
Id                       Owner      Submitted   ST PRI Class        Running On
------------------------ ---------- ----------- -- --- ------------ -----------
f18n13s.290.0            shecar      7/21 09:47 R  50  No_Class     f01n01s
f18n13s.280.0            shecar      7/21 09:21 I  50  background
f18n13s.281.0            shecar      7/21 09:21 I  50  parallel
f18n13s.282.0            shecar      7/21 09:21 I  50  challenge
```

The output for the waiting jobs using `llbeans -w` is described in Example 8-5.

**Example 8-5 llbean output for waiting jobs**

```
[shecar@f18n13e](/u/home/shecar/JoeS)$ llbeans -w
Job: f18n13s.281.0 blitzen 0 07/21/00-09:21:15 05:00:05 16 shecar parallel
Job: f18n13s.282.0 zippy 0 07/21/00-09:21:26 05:00:05 16 shecar challenge
Job: f18n13s.280.0 pokey 0 07/21/00-09:21:05 05:00:05 16 shecar background
```
Note that for waiting jobs, this field is the Wallclock Estimate.

In Example 8-6, `llbeans -r` shows all the running jobs.

**Example 8-6  llbean output for running jobs**

```bash
[ shecar@f18n13e ]$ llbeans -r
Job: f18n13s.290.0 wuzza 3 07/21/00-09:47:22 07/21/00-20:47:27 16 shecar No_Class
```

And `llbeans -r -v` shows running jobs and associated nodes:

```bash
[ shecar@f18n13e ]$ llbeans -r -v
Job: f18n13s.290.0 wuzza 3 07/21/00-09:47:22 07/21/00-20:47:27 16 shecar No_Class
Nodes: f01n01s f01n02s f01n03s f01n04s
```

**LL_get_nodes**

`LL_get_nodes` is a C code that queries the LL data API. `LL_get_nodes` grabs information on the nodes and returns it in the format described in Example 8-7.

Note that the running job (job_name = wuzza) is using the four nodes f01n02s, f01n05s, f01n08s, and f01n10s.

**Example 8-7  Sample LL_get_nodes output**

```text
f01n02s.navo.hpc.mil:4096;:4:0:Idle:
f01n05s.navo.hpc.mil:4096;:4:0:Idle:
f01n08s.navo.hpc.mil:4096;:4:0:Idle:
f01n10s.navo.hpc.mil:4096;:4:0:Idle:
f01n02s.navo.hpc.mil:4096;:4:4:Busy:f18n13s.navo.hpc.mil.295.0,
f01n05s.navo.hpc.mil:4096;:4:4:Busy:f18n13s.navo.hpc.mil.295.0,
f01n08s.navo.hpc.mil:4096;:4:4:Busy:f18n13s.navo.hpc.mil.295.0,
f01n10s.navo.hpc.mil:4096;:4:4:Busy:f18n13s.navo.hpc.mil.295.0,
```

<table>
<thead>
<tr>
<th>node name</th>
<th>physical mem</th>
<th>Node state</th>
<th>LL job id for job(s) running on the node</th>
<th>task slots in use</th>
</tr>
</thead>
</table>
The nodes are returned one node per line and the fields are : (colon) separated. The job queue, corresponding to the previous example, is shown in Example 8-8.

Example 8-8   llbeans alternate use

```
[shecar@f18n13e]/u/home/shecar/JoeS]$ llbeans -r -v
Job: f18n13s.295.0 wuzz 3 07/21/00-10:21:45 07/21/00-21:21:50 16 shecar No_Class
Nodes: f01n02s f01n05s f01n08s f01n10s
```

8.2 External schedulers

In this section, we provide an overview of the external schedulers, which are developed by customers using the LoadLeveler API. The objective of this section is to provide an overview of these schedulers and their functionality. Please refer to the documentation for the respective schedulers for detailed information.

8.2.1 Maui

The Maui scheduler was originally designed and written at the Maui High Performance Computing Center (MHPCC) to address limitations in existing resource managers of the time, such as low system utilization and limited configuration for job prioritization. It is now developed and maintain by the Supercluster Research and Development Group (http://supercluster.org).

Here we will outline some of the key features of the Maui scheduler. A more detailed and up-to-date description, including instructions for installation and use, can be found at:

http://supercluster.org/maui

Overview

While LoadLeveler is still responsible for managing resources, it is the Maui scheduler that decides which nodes to use and the start time of the jobs. As with other external schedulers, Maui functions by using the scheduling API of LoadLeveler by querying the negotiator via the data API to retrieve job and node information. This information is then run against one or more scheduling decisions, known as policies, to decide whenever the job will run, the following policies are available:

- Job prioritization
- Backfill
- Reservations
- Throttling policies
Backfill
Backfill is now used by many schedulers to improve job throughput and system utilization by allowing the scheduler to run lower priority jobs with available resources as long as the highest priority queued job is not delayed in running. As with Backfill, the Maui scheduler expects a submitted job to have an associated wallclock time. This enables the scheduler to know when a running job will complete at the latest, allowing a look ahead to be performed.

Maui orders the job queue by priority. Each job in the list is started in turn and for each a reservation is created when it starts. This continues until a job is encountered that cannot be started. At this point, a job reservation is created for the highest priority queued job so as to prevent backfilled jobs from delaying the highest priority job from running.

By default, Maui reserves only the highest priority job, and will not protect other priority jobs from being delayed by backfilled jobs. This can be modified with the RESERVATIONDEPTH parameter instructing how far down the list to create reservations, allowing high priority work to run on time.

The scheduler now attempts to run jobs that will fit in available resource gaps without delaying reserved priority jobs. The parameter BACKFILLMETRIC allows specification of the criteria by which backfill job 'suitability' is evaluated, defaulting to running the largest possible jobs on the available resources. This is possible because all jobs and reservations have a starttime and a wallclock time. Hence the completion time of jobs is known but only known accurately if the estimated wallclock is accurate. Inaccuracies in wallclock time will lead to jobs completing early and creating idle resource windows available for backfill.

Maui performs two backfill scheduling passes. For each pass, a list of jobs are generated that are considered eligible for backfill according to any defined throttling policies, such as preventing one user from dominating the system. On the first sweep are those jobs which meet the requirements of the scheduled soft fairness throttling policy. On the second sweep, any jobs which meet the requirements of the hard fairness throttling policy are added to the list.

The possible backfill algorithms are:

**FIRSTFIT** Each entry in the list of candidate jobs is considered for backfill starting with the highest priority idle job. This is the default if no algorithm is specified in the configuration file.
**BESTFIT**

From the list of eligible jobs, those that would fit in the current backfill window are considered for dispatch.

Each job is determined for its "degree of fit" according to the parameter BACKFILLMETRIC, which can evaluate jobs based on various measures of job duration and size.

The job that has the best "degree of fit" is started and the backfill window size is adjusted accordingly.

The list continues to be processed while there are backfill jobs and free resources are available.

**GREEDY**

Greedy behaves like BESTFIT but rather than attempting to locate and start the best jobs one at a time, it attempts to determine the best possible combination of jobs which will meet the criteria specified by the BACKFILLMETRIC parameter.

**PREEMPT**

This algorithm takes advantage of Loadleveler's ability to requeue active jobs. Under this policy, rather than maintaining priority job reservations, backfill jobs are optimistically started even if they may potentially conflict with higher priority blocked idle jobs. These backfill jobs are considered PREEMPT jobs, allowing priority jobs to stop them and use their resources as soon as aggregate idle and preemptible resources become available. When these resource become available, Maui will determine the minimal impact list of backfill jobs to preempt, will preempt them, and will then start the priority job on the needed resources. This policy overcomes many of the drawback of other backfill methods where job wallclock inaccuracies cause scheduling inefficiencies.

**Job prioritization**

Deciding on which jobs run first is one of priority. Queued jobs are prioritized based on a value derived from factors such as the job's queue duration, wallclock time, the user's historical system usage, or even the number of times a job has been bypassed by a lower priority job.

**Service factor**

Favors jobs based on current job queue time, expansion factor, backfill bypass count, and/or other job service metrics.

**Resource factor**

Favors value based on the requested resource for a job, such as processors, memory, swap, walltime, processor-seconds, and so on.
**Fairshare factor**  
Historical percentage of system utilization of job's user, group, account, or Quality of Service (QoS).

**Credential factor**  
Assignment of specific additional priority to users, groups, accounts, and classes.

**Target factor**  
Grows exponentially over time as the service target nears (that is, target = QueueTime < 2.0 hours). This causes Maui to attempt to make sure the job reaches its scheduled target.

**Throttling policies**

One of the many problems with scheduling is to prevent a particular job from dominating the system resources. Maui's ability to restrict the instantaneous resource consumption is controlled by a set of related policies. The purpose is to prevent the system from being dominated by an individual user, group, account, or QoS. They are similar in operation to filter; either allowing or disallowing jobs for scheduling eligibility based on the system usage at that point in time. Limits can be applied per user, group, account, QoS, or queue with the configuration keywords below.

- **MAXJOB**: Jobs running
- **MAXJOBQUEUED**: Jobs queued (not non-queued)
- **MAXPROC**: Number of processors
- **MAXPS**: Processor seconds
- **MAXPE**: Processor equivalent
- **MAXWC**: Wallclock time
- **MAXNODE**: Nodes
- **MAXMEM**: Memory

In order to restrict the user "kannan" to a total of 2 jobs and 16 processors at any time, the following is set in the maui.cfg:

```
USERCFG[kannan] MAXJOB=2 MAXPROC=16
```

If flexibility is required on the limits, this can be applied with hard and soft limits. Hard limits cannot be exceeded in any situation, while soft limits can be exceeded only if no other jobs can run. In the statement below, we define the default limits for all users:

```
USERCFG[DEFAULT] MAXJOB=2,4 MAXPROC=16,24
```

Individual users or groups can be altered to have different limits to the default. Here the user peterm has a job hard and softlimit of 4 and 8 respectively, with a maximum processor hard and softlimit of 48 and 64:
USERCFG[peterm] MAXJOB=4,8 MAXPROC=48,64

If a user submits a job and exceeds their policy limits, then the job will be placed in the non-queued list in a deferred state until the job is able to join the normal job queue. The job cannot run in partition DEFAULT (rejected by policy MaxJobPerUser: policylevel: "SOFT"). Setting a default limit for all groups is applied similarly. In this case, all groups can only consume up to 128 nodes in total at any point in time:

GROUPCFG[DEFAULT] MAXNODE=128

In addition to limits on the jobs that can be simultaneously run, Maui also allows a full suite of constraints on which idle jobs can be considered for scheduling. This is done in the same format as active job limits with the addition of the letter 'I' (for idle) into the specific limit (that is, MAXJOB or MAXPROC).

**Node allocation**

Which "free" nodes are to be allocated to a job is determined by the node NODEALLOCATIONPOLICY keyword. The possible options are:

- MINRESOURCE - Nodes with the minimum configured set of resources.
- CPULOAD - Nodes with the lowest CPU load average.
- LASTAVALIABLE - The earliest start time for a job is used to select the idle resources available at that time, while not reserving currently idle nodes, so backfill can be more effective; hence, it is more likely that this will be used.
- NODEPRIORITY - A site defined priority.
- FASTEST - Allocate the fastest available nodes to the job.

**Fairshare**

Fairshare generally is a method of allocated equal shares of a value. When applied to a batch queue, however, it becomes a priority issue. Maui attempts to achieve fairshare by adjusting, but NOT enforcing, a job's priority based on the historical system usage of the job's user, group, account, or QoS. Fairshare has a target value. The method by which resource allocation is used in a fairshare measurement is achieved with the FSPOLICY keyword. The options FSINTERVAL, FSDEPTH, and FSPOLICY dictate the value of the historical usage to be valued. In this example the system will monitor processor seconds over a 2 week period. For example, to monitor the processor seconds for a two week period, the following settings can be used:

- FSPOLICYDEDICATEDPS
- FSDEPTH14
- FSINTERVAL24:00:00
- UserCfg[DEFAULT]FSTARGET=10
By default, all users will receive a target of 10 percent of the delivered processor seconds. The exception here is that kannan will receive a floor of 50 percent and peterm will be limited to 25 percent, the group itso will also receive a ceiling of 33 percent. What this means is that Maui will adjust job priorities in an attempt to keep each default user using approximately 10 percent of delivered processor seconds, kannan, using at least 50 percent, and peterm and itso using less than 25 and 33 percent of delivered process seconds respectively.

Fairshare therefore enables a site to influence the workload; if monitored and modified to react to the ever-changing demand of user requirements, job throughput may be improved.

Maui provides interfaces to allocation management tools such as QBank, which further enhance the ability to handle fair distribution of compute cycles over medium and long-term time frames.

**Quality of Service (QoS)**

This facility allows for the special consideration of classes of jobs, users, groups, and so on. A QoS object can be thought of as a container that holds privileges, such as job prioritization, policy exemptions, and resource access.

The object itself has an access control list that describes who can access the privileges. You may configure different QoSs with their own priorities, exemption, access, and resource settings. You may also define user, group, account, and class access, of which a job can be assigned a default QoS.

At job submission, you can request a QoS; otherwise, the default setting will be used. If you decide to modify the QoS after job submission, you can adjust the QoS within your allowed QoS range. The administrator can set the QoS of any job to any limit.

Jobs are permitted access to a QoS's set of privileges with the QoS default (QDEF) or QoS access list (QLIST) settings in the fairshare configuration file fs.cfg.

**Reservations**

An advance reservation is the method by of allocating a block of resource for a set period of time. A reservation has three properties

- A resource list
A start time
A specific amount of time

This is set against an access list. It is the responsibility of the scheduler to enforce the access list for the lifetime of the reservation.

Two types of advance reservations are available:

- Administrative
- Standing

Administrative reservations are ideal for situations that are unpredictable in nature. An example would be the need to create a reservation on all or sets of nodes for an infinite period for system maintenance. For example, to create a reservation on all or sets of nodes for an infinite period for system maintenance, use the following command:

```
$ setres -s 14:00:00_05_29/01 ALL
reservation created
reservation 'SYSTEM.1' created on 8 nodes (8 tasks)
sp6n14.msc.itso.ibm.com:1
sp6n13.msc.itso.ibm.com:1
sp6n12.msc.itso.ibm.com:1
sp6n11.msc.itso.ibm.com:1
sp6n10.msc.itso.ibm.com:1
sp6n09.msc.itso.ibm.com:1
sp6n08.msc.itso.ibm.com:1
sp6n07.msc.itso.ibm.com:1
```

The reason for the system reservation may be to initiate a system reboot when the time for a return to service is unknown. The situation might be that the administrator may want jobs to be scheduled and dispatched until the system has passed post upgrade checks. Individual nodes or groups of node can also be targeted; maybe you need to verify the procedures for applying a software patch on an "isolated" node.

Another common purpose is to allow a list or group of users or user groups to have exclusive access to a number of nodes for limited time period, such as a 24 hour training period or for running benchmark suites. Here the reservation would created on that day at 7:30 AM for the group training on nodes sp6n12 and sp6n13:

```
$ setres -g training -s 07:30 -d +24:00:00 'sp6n12 sp6n13'
```

To set the reservation for June 2 at 7:30 AM, the syntax would be:

```
$ setres -g training -s 07:30_06/02 -d +24:00:00 'sp6n12 sp6n13'
```
If the reservation period is unknown, then do not set a duration time, and it will default to infinity.

If use of the reservation has finished before the reservation expires, it can be removed with the `releaseres` command, returning the resources for immediate use:

```
releaseres <reservation_id>
```

Usually job access to reservations is automatically handled based on the user, group, or job attributes and needs no prior user knowledge. However, a user may want to specify the reservation in which to run (access controls permitting). The user notifies Maui, via the LoadLeveler command file variable `# @ comment`, which reservation to use:

```
# @ comment = advres[=<reservation_id>]
```

Standing reservations are a method of repetitively reserving resources for a constant set time period. The standing reservations are defined in the `maui.cfg` file and require a scheduler reconfiguration for new or modified reservations to take effect. An example of a standing reservation may be a requirement to reserve nodes to serve interactive login sessions, Monday to Friday between the hours of 7:30 AM and 7 PM. Outside of these times, the nodes would be available to run jobs. Jobs would only be allocated to nodes that would complete before 7:30 AM on each weekday. To do this, we set these values to these configuration parameters:

```
SRNAME[1]    ISS
SRDAYS[1]    Mon Tue Wed Thu Fri
SRSTARTTIME[1]    07:30:00
SRENDTIME[1]    18:30:00
SRDEPTH[1]    4
SRHOSTLIST[1]    sp6n13 sp6n14
SRTASKCOUNT[1]    2
```

The login sessions could be configured via a Domain Name Service (DNS) round-robin method or with a load balancing package, such as IBM WebSphere, which employs a configurable rotation metric.

In the standing reservation below, nodes sp6n01 and sp6n03 are allocated between the hours of 7:30 AM to 6:30 PM to run only jobs for the interactive parallel class `inter_class`:

```
SRNAME[2]    POE
SRDAYS[2]    Mon Tue Wed Thu Fri
SRSTARTTIME[2]    07:30:00
SRENDTIME[2]    18:30:00
SRDEPTH[2]    4
SRHOSTLIST[2]    sp6n01 sp6n03
```
srtaskcount[2]        2
srclasslist[2]        inter_class
sruserlist[2]         ALL

sp4n05 is allocated to run jobs to the maximum wallclock limit of 10 minutes. If
the node has reached its allocation, then the job will be allocated to an idle node
in the global node pool if one is available. The following configuration settings are
required to enable this function:

srname[0]             SHORTPOOL
srdays[0]             Mon Tue Wed Thu Fri
s startime[0]         07:30:00
s rendtime[0]          18:30:00
s depth[0]            10
srhostlist[0]         sp4n05
srmaxtime[0]          600
sruserlist[0]         ALL
srresource[0]         PROCS=2;MEM=512
srtaskcount[0]        1

This approach provides flexibility, allowing a resource to be dedicated to
potentially fast job turnaround but at the expense of utilization. If the system is
full, and a queue of jobs exist whose wallclock limit is greater than 10 minutes,
then the node will remain idle until the reservation is removed. If the any nodes
quoted in SRHOSTLIST are SMP nodes, then the SRTASKCOUNT should be
the total processor count for SRHOSTLIST.

Information on existing reservations can be viewed with the showres command.
This command is restricted to administrative users listed in the maui.cfg file.

Reservations
Information on existing reservations can be viewed with the showres command.
This command is restricted to administrative users listed in the maui.cfg file:

<table>
<thead>
<tr>
<th>Type</th>
<th>ReservationID</th>
<th>S</th>
<th>Start</th>
<th>End</th>
<th>Duration</th>
<th>N/P</th>
<th>StartTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>SHORTPOOL.0</td>
<td>-</td>
<td>-0:01:00</td>
<td>8:11:54</td>
<td>8:12:54</td>
<td>2/2</td>
<td>Tue May 29 10:17:06</td>
</tr>
<tr>
<td>User</td>
<td>SHORTPOOL.1</td>
<td>-</td>
<td>21:11:54</td>
<td>1:08:11:54</td>
<td>11:00:00</td>
<td>2/2</td>
<td>Wed May 30 07:30:00</td>
</tr>
<tr>
<td>User</td>
<td>SHORTPOOL.2</td>
<td>-</td>
<td>1:21:11:54</td>
<td>2:08:11:54</td>
<td>11:00:00</td>
<td>2/2</td>
<td>Thu May 31 07:30:00</td>
</tr>
<tr>
<td>User</td>
<td>SHORTPOOL.3</td>
<td>-</td>
<td>2:21:11:54</td>
<td>3:08:11:54</td>
<td>11:00:00</td>
<td>2/2</td>
<td>Fri Jun 1 07:30:00</td>
</tr>
<tr>
<td>User</td>
<td>ISS.0</td>
<td>-</td>
<td>-1:28:00</td>
<td>8:11:54</td>
<td>9:39:54</td>
<td>2/2</td>
<td>Tue May 29 08:50:06</td>
</tr>
<tr>
<td>User</td>
<td>ISS.1</td>
<td>-</td>
<td>21:11:54</td>
<td>1:08:11:54</td>
<td>11:00:00</td>
<td>2/2</td>
<td>Wed May 30 07:30:00</td>
</tr>
<tr>
<td>User</td>
<td>ISS.2</td>
<td>-</td>
<td>1:21:11:54</td>
<td>2:08:11:54</td>
<td>11:00:00</td>
<td>2/2</td>
<td>Thu May 31 07:30:00</td>
</tr>
<tr>
<td>User</td>
<td>ISS.3</td>
<td>-</td>
<td>2:21:11:54</td>
<td>3:08:11:54</td>
<td>11:00:00</td>
<td>2/2</td>
<td>Fri Jun 1 07:30:00</td>
</tr>
<tr>
<td>User</td>
<td>POE.0</td>
<td>-</td>
<td>-1:28:00</td>
<td>8:11:54</td>
<td>9:39:54</td>
<td>1/1</td>
<td>Tue May 29 08:50:06</td>
</tr>
<tr>
<td>User</td>
<td>POE.1</td>
<td>-</td>
<td>21:11:54</td>
<td>1:08:11:54</td>
<td>11:00:00</td>
<td>1/1</td>
<td>Wed May 30 07:30:00</td>
</tr>
<tr>
<td>User</td>
<td>POE.2</td>
<td>-</td>
<td>1:21:11:54</td>
<td>2:08:11:54</td>
<td>11:00:00</td>
<td>1/1</td>
<td>Thu May 31 07:30:00</td>
</tr>
<tr>
<td>User</td>
<td>POE.3</td>
<td>-</td>
<td>2:21:11:54</td>
<td>3:08:11:54</td>
<td>11:00:00</td>
<td>1/1</td>
<td>Fri Jun 1 07:30:00</td>
</tr>
</tbody>
</table>

13 reservations located
Currently available backfill
Maui can provide an estimated number of processors available to a user with the `showbf` command:

```
$ showbf
backfill window (user: 'markr' group: 'staff' partition: ALL) Tue May 29
13:31:10
no procs available
```

The output above shows that no processors are available; this is correct because the system reservation SYSTEM.1 is still in force, due to its infinite time period, as shown in the `showres` command output:

```
User   SYSTEM.1     -   -0:01:00    INFINITY  INFINITY  8/14 Tue May 29 10:17:06
Job Queue
```

One difference between Maui and LoadLeveler is the queue principle. In LoadLeveler a single queue shows all jobs regardless of their state. Maui divides the queue into active, idle, and non-queued. Here the important feature is that the non-queued jobs are not considered in the scheduling algorithm until they return to the idle queue.

Statistics
The `showstats` command allows the administrator to investigate various accounting statistics for the system. The information reported is valid for the lifetime of the current scheduler and is reset upon start-up or with the `resetstats` command. The `showgrid` command reports various measurements from this data, such as average queue time, wallclock accuracy, and efficiency of jobs. Information for nodes, groups, and users can also be obtained with the `showstats` command; with no parameters passed, it will give a summary of the statistics for the current server duration:

Eligible/Idle Jobs: 65/69 (94.203%)
Active Jobs: 21
Successful/Completed Jobs: 1829/1829 (100.000%)
Avg/Max QTime (Hours): 1.82/75.36
Avg/Max XFactor: 0.28/76.67
Dedicated/Total ProcHours: 19885.62/34858.28 (57.047%)
Current Active/Total Procs: 161/208 (77.404%)
Avg WallClock Accuracy: 33.591%
Avg Job Proc Efficiency: 95.842%
Est/Avg Backlog (Hours): 15.13/17.32
Maui also generates statistical information daily; these flat files have the format <day_month_date_year>, which is written to the stats directory specified with the STATDIR parameter specified in the maui.cfg file. These files can be used as an input to the simulator and for analyzing historical trace files of system usage with the profiler facility.

**Testing and simulation**

One of the unique features is the test and simulation modes. When using either the TEST or SIMULATION mode, they can be run while a production scheduler is still running the current configuration.

The test mode allows you to practice modifying policies without impacting the running server. However, you will need to have separate Maui installations, including configuration files that use a separate port, data directories, and policy parameters.

The simulator can take input from a workload from an active scheduler allowing you to analyze how the current workload would be scheduled under the different configurations. While the simulation is in progress, all admin commands will function and return information for the workload or even allow for the submission or cancellation of jobs, though this would affect the result of the simulation.

Both modes allow an administrator to test and evaluate altered configurations safely before deploying the modifications in a production environment. This approach prevents situations where administrators react to issues that adversely affect work flow that are caused by an untested configuration.

**Troubleshooting**

As with LoadLeveler, there are logging capabilities in Maui. These are controlled via the maui.cfg file:

```
LOGDIR /home/loadl/maui/log
LOGFILE maui.log
LOGLEVEL 3
```

Care should be exercised at increasing the log level to a high value when running routinely, as high levels will generate a huge amount of data. It is more useful to increase the log level only when attempting to trace a problem.

Your first action during problem determination should be to examine the maui.log file for the progress of scheduling and allocation steps. If all appears to be functioning normally, then move on to the LoadLeveler logs to trace the job through the various LoadLeveler steps. Determining why a job has failed to start
or why it has been placed in the non-queued queue with a deferred status can be analyzed with the `checkjob` command. The output below indicates that the class the job was submitted with may be at fault, which has been highlighted for clarity purposes:

```
$ checkjob sp6en0.176.0
checking job sp6en0.176.0
State: Idle  (User: markr  Group: staff  Account: [NONE])
WallTime: 0:00:00  (Limit: 0:30:00)
QueueTime: Mon May 28 16:50:55
Total Tasks: 1
Req[0] TaskCount: 1 Partition: ALL
Network: [NONE] Memory >= 0 Disk >= 0 Features: [NONE]
Opsys: AIX51  Arch: R6000  Class: [serial] 1
IWD: /u/markr/CHAPTER9/MAUI  Executable: /u/markr/CRUNCH/a.out
QOS: 0  Bypass: 0  StartCount: 0
Partition Mask: ALL
PE: 1.00  Priority: 1810
cannot select job sp6en0.176.0 for partition DEFAULT (Class)
```

Additional commands, such as `diagnose`, also provide hints on why a job may be failing to start.

**Predicted job start time**

Because the scheduler knows when running jobs will complete at the latest, due to the wallclock time specified for the job, and when those resources can be released, it can provide an estimated start time for a job based on the current backfill priority:

```
showstart <jobid>
```

This might change however if higher priority jobs enter the system or if the job is bypassed in favor of another job.

**Scheduler and internal checks**

The `diagnose` command provides the administrator with information about scheduling and requests that Maui perform various internal checks. The areas available for reporting include:

- Accounts
- Fairshare
- Groups
- Job
- Frames
- Nodes
- Priority
Advantages
- Reservations: The administrator can set a system downtime in the future to perform patch installation or a system reboot without interrupting current work.
- Simulator: Provides a method of profiling historical or current workload on new configurations before deploying onto a production system.
- User commands: Extensive information, such as the predicted start time of a job.
- Admin commands: Extensive, flexible and powerful.
- Priority: Fine granularity.
- Support for consumable resources.

Limitations
- Configuration: Fine tuning of parameters may be time consuming, but this depends on the complexity of your policy and the time spent testing.
- Availability: No backup. If the Maui server goes down, then jobs will not be scheduled until it resumes service.

8.2.2 Special Resource Scheduler (SRS)

The Special Resource Scheduler (SRS) was developed at Cornell University in 1996 for use on the 512 node IBM SP2 parallel machine. Cornell began using the SP2 with LoadLeveler and it soon became apparent that a backfill algorithm was necessary. In 1995/1996, LoadLeveler had not yet incorporated an internal backfill algorithm, so researchers at IBM and Cornell jointly developed the first LL schedule interface.

Attached to this interface was Easy-LL, a backfill scheduler. This was the first implementation of the external LL scheduler interface. Easy-LL at Cornell was the first external scheduler to be used in production. Both Easy-LL and the external scheduling interface worked well, but did not allow for the use of heterogeneous node resources. Because the SP2 at Cornell was a heterogeneous system, there was a need for a different scheduler. SRS was developed to fill this need.
Using the Easy-LL scheduler as a starting point, enhancements were implemented to SRS to allow jobs to specify node resources and for the scheduler to select these resources by using the context of the backfill algorithm. From the original algorithm, there are three basic steps involved in scheduling a job:

1. Determine if the job on the top of the queue can run on currently idle nodes.
2. If it cannot, determine when it can run. When will resources become available to run the job? This time is called the Shadow Time (ST).
3. Look through other jobs on the queue. If jobs can be run that execute on free nodes (or less) for ST (or less) time, then start this job. The execution of this job will not interfere with the start of the top job on the queue; therefore, the job can be successfully backfilled.

The original scheduler treated all nodes as a simple resource that was identical to all other nodes and could either be idle or busy. So, the accounting of node resources could be done with an integer count. For example, there may be four nodes free and the top job needs six to run. In this case, a variable such as FreeNodes=4 might be used by the backfill scheduler. Note that these could be any four nodes. To the scheduler, they simply represent free slots where processes of a parallel job might run.

In order to expand this basic logic for use by SRS, the accounting of resources was changed from a simple integer to a list. So, FreeNodes=4 might become a list of four nodes and their associated resources. With this change, each of the three backfill steps could be expanded to include resource computations.

The three basic steps now become:

- See if the job on the top of the queue can run on currently idle nodes, matching resources requested by the user with node resources.
- If it cannot, determine when it can run. When will resources become available to run the job? Look for those nodes that will become free in the future AND match the resources requested by the user. This time is called the Shadow Time (ST).
- Look through other jobs on the queue. If jobs can be run that execute on free nodes (or fewer) for ST (or less) time, then start this job. In order for the jobs to run, user requested resources must match node resources.

The expansion of this logic implies two sweeping changes for the original algorithm. First, the scheduler must be expanded so that internal structures now take resource information into account. Second, a method needs to be developed that will allow the users to specify a list of resources required by their job.
User resource specification was added by using the comment field in the LL command file. A grammar was created using the following format:

```
@#comment =
N@RESOURCE1=entry1,RESOURCE2=entry2;N@RESOURCE3=entry3
```

Where:
- `@#comment =` is the LL command file descriptor for the comment field.
- Different resource specifications are separated with `;`.
- `N@` specifies the number of nodes you want with the designated resource type.
- `RESOURCE1`, `RESOURCE2`, and `RESOURCE3` can be `MEM` and `TYPE`. These resource designations need to be in ALL CAPS.

Some examples of this grammar in use are:
- `@#comment = 2@TYPE=wide,MEM=1024 ; TYPE=blue`
  This specification requests two wide nodes with 1024 MB of memory (each!) and a node designated as a blue node by the system administrator. The first 1024 MB node discovered by the scheduler will be the master node.
- `@#comment = 1@MEM=2048 ; 2@TYPE=wide ; 4@TYPE=thin ; TYPE=private`
  This line requests one node with 2048 MB, two wide nodes, four thin nodes, and a node marked as private. Since the 2048 MB node is requested first, it will be selected as the master node.

The user can specify as many distinct resources as there are nodes in the system. The user can also (and usually will) specify fewer resources than there are nodes in the system. In this case, SRS will assign arbitrary nodes, based on earliest availability time, to the number of remaining nodes.

SRS manages two types of resources; those that LL maintains (such as memory) and resources set by the LL system administrator. In the examples above, the resources set by the administrator are designated by the TYPE keyword (thin/wide and node color, in these examples). The administrator-designated resources are identified using the feature keyword within a local configuration file for a node. The administrator places the desired feature in this field. SRS extracts this field when it encounters it. If a user requests the feature using a TYPE statement in the comment line, SRS will apply resource checking as for all other existing features.

Some notes about specifying resources:
The specification statements for a node set can occur in any order. For example, \(2@\text{MEM}=256,\text{TYPE}=\text{wide}\) is equivalent to \(2@\text{TYPE}=\text{wide},\text{MEM}=256\).

It is permissible to leave off the \(1\) from the resource specification if you only need a single node.

Separate unique node specifications are set with a \(;\) (semicolon).

Separate multiple resources for the same node are set with a \(,\) (comma)

Separate multiple \text{TYPE} entries are set with a \(_\) (underscore).

Specifying no resource requirements will assign a generic node, which is defined by the system administrator within the scheduler. Specifying a single resource (for example, \text{MEM}=256) will assign the first available node matching the resource to the job. In this example, the 256 MB node might be either wide or thin.

Only one comment line is allowed per command file. Therefore, you need to place all resource requests on this single line.

The following example illustrates the operation of SRS. Consider the job queue given Figure 8-1.

<table>
<thead>
<tr>
<th>Job ID</th>
<th>nodes</th>
<th>hours</th>
<th>resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job 1</td>
<td>4</td>
<td>12</td>
<td>(4@\text{MEM}&gt;=512)</td>
</tr>
<tr>
<td>Job 2</td>
<td>8</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Job 3</td>
<td>4</td>
<td>12</td>
<td>(1@\text{MEM}&gt;=2048)</td>
</tr>
<tr>
<td>Job 4</td>
<td>6</td>
<td>12</td>
<td>(6@\text{wide})</td>
</tr>
<tr>
<td>Job 5</td>
<td>5</td>
<td>2</td>
<td>(1@\text{sw1})</td>
</tr>
<tr>
<td>Job 6</td>
<td>2</td>
<td>3</td>
<td>(2@\text{wide})</td>
</tr>
<tr>
<td>Job 7</td>
<td>1</td>
<td>4</td>
<td>(1@\text{wide})</td>
</tr>
</tbody>
</table>

*Figure 8-1  Job queue*

The jobs request a variety of special resources, including different physical memory amounts, wide or thin nodes, and various node-locked software packages. The example job queue is submitted to the following 24 node SP consisting of eight wide nodes and 16 thin nodes. (see Figure 8-2).
In this example, the machine has eight wide nodes with the following memory distribution: Three 512 MB, four 1024 MB, and one 2048 MB nodes. There are also 16 thin nodes each with 256 MB of memory. Four of the thin nodes have a node locked license for the sw2 software package. Two of these nodes also have licenses for the node locked software package sw1. If the nodes of the machine are initially free, SRS will start as many nodes as possible on available nodes. The jobs are examined in submit time order and requested resources are also considered. In the example, Jobs 1, 2, and 3 will all start immediately, as shown in Figure 8-3. Job 1 receives four nodes, three with 512 M, and one with 1024 MB to satisfy the request for four nodes with 512 MB (or greater) of memory. Job 2, which requests no special resources, starts on eight thin nodes (SRS will attempt to assign nodes with the least available special resources to jobs with no special requests). Job 3 requests four nodes, one with 2048 MB of memory. Hence, Job 3 receives three thin nodes and the single wide node with 2048 MB of memory. (see Figure 8-4 on page 138).
Next, Job 4 requests six wide nodes in order to run. Since there are only three wide nodes available, SRS does not start Job 4. Instead, SRS computes the possible starting time for Job 4, which is 12 hours from now, when Job 1 finishes. In addition, SRS protects the six nodes it will use in the future to run Job 4. SRS selects the four nodes currently running Job 1, and two of the currently free wide nodes. SRS next looks at Job 5 and assigns it the appropriate thin nodes. Job 5 requests and obtains one node with node locked software sw1. Since the other nodes of Job 5 are unspecified, the remaining thin nodes are assigned (including those with node locked software) rather than using the wide nodes (see Figure 8-4 on page 138).
SRS next looks at Job 6, which requires two wide nodes. Job 6 is assigned one free wide node and one wide node protected for Job 4. Since Job 6 will complete within four hours, it will not impact the start of Job 4, 12 hours from now. Therefore, Job 6 can be safely backfilled on any node reserved for Job 4. Finally, SRS starts Job 7 on the remaining free wide node. This node is reserved for Job 4 but Job 7 will complete in four hours and will not impact the start time of Job 4. The final job mapping is shown in Figure 8-5.
Resource usage and monitoring

In this chapter, we describe methods for resource accounting used by LoadLeveler during job allocation, aiding the understanding of how the cluster is being utilized and improving cluster configurations.

The monitoring of logs provides a wealth and information for problem diagnosis of both work and internal cluster operations. The following topics are discussed:

- Accounting
- Monitoring
- Tools for monitoring
- Mail notification
- Managing log files
- Troubleshooting tips
9.1 Accounting

Accounting is sometimes considered a necessary evil by managers and administrators alike. But it can provide a valuable insight into what people are doing and how much of the machine resources they are consuming.

History files
Fundamental to LoadLeveler accounting are the history files. LoadLeveler stores the accounting files in the history files in the local directory on the node when accounting is enabled. The administrator can specify the history file using the HISTORY keyword. There are two types of history files:

- **Local history**: The history file, which is local to the scheduling machine.
- **Global history**: The local history files from machines in the cluster merged together into a single history file. These are stored in the GLOBAL_HISTORY directory, which is specified in the LoadL_config file:
  
  \[
  \text{GLOBAL_HISTORY} = $(SPOOL)
  \]

Accounting settings
LoadLeveler accounting enablement and the mode of operation is achieved with the following keyword in the LoadL_config file:

\[
\text{ACCT} = <\text{settings}>
\]

The possible settings for ACCT are as follows:

- **A_OFF**: Enables accounting. This is the default.
- **A_ON**: Enables accounting.
- **A_VALIDATE**: Enables validation of LoadLeveler account numbers.
- **A_DETAIL**: Enables extended accounting, recording detailed resource consumption by machine and event for each job step.

Accounting data update
The frequency with which the startd daemon informs the schedd daemon of accounting data for running jobs is controlled with JOB_ACCT_Q_POLICY:

\[
\text{JOB_ACCT_Q_POLICY} = <\text{seconds}>
\]

The lesser value of JOB_ACCT_Q_POLICY or JOB_LIMIT_POLICY is used to set the interval for the startd daemon to collect resource usage for running jobs and also on how often the job_cpu_limit is checked.
Note that a lower value for JOB_ACCT_Q_Policy will increase network traffic:

\[
\text{JOB\_LIMIT\_POLICY} = \text{<seconds>}
\]

If no value is set for JOB\_LIMIT\_POLICY, then the value of POLLING\_FREQUENCY \* POLLS\_PER\_UPDATE is used.

If a job starts and completes within the interval, you will not see any consumed resource reported. To improve the update of the job information, you can decrease the JOB\_LIMIT\_POLICY value.

An example of the \texttt{llq} extended options to display usage is shown below with the CPU time for each running job displayed:

\[
\text{\$ llq -x}
\]

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On Job</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4n01.1946.0</td>
<td>markr</td>
<td>6/2 15:45</td>
<td>I</td>
<td>50</td>
<td>serial_l</td>
<td>none</td>
<td>record</td>
</tr>
<tr>
<td>sp4n01.1945.0</td>
<td>markr</td>
<td>6/2 15:45</td>
<td>R</td>
<td>50</td>
<td>serial_l</td>
<td>sp4n01</td>
<td>00:05:21</td>
</tr>
<tr>
<td>sp4n01.1944.0</td>
<td>markr</td>
<td>6/2 15:45</td>
<td>R</td>
<td>50</td>
<td>serial_l</td>
<td>sp4n01</td>
<td>00:05:21</td>
</tr>
<tr>
<td>sp4n01.1943.0</td>
<td>markr</td>
<td>6/2 15:45</td>
<td>R</td>
<td>50</td>
<td>serial_l</td>
<td>sp4n01</td>
<td>00:05:23</td>
</tr>
<tr>
<td>sp4n01.1942.0</td>
<td>markr</td>
<td>6/2 15:45</td>
<td>R</td>
<td>50</td>
<td>serial_l</td>
<td>sp4n01</td>
<td>00:05:23</td>
</tr>
<tr>
<td>sp4n01.1941.0</td>
<td>markr</td>
<td>6/2 15:45</td>
<td>R</td>
<td>50</td>
<td>serial_l</td>
<td>sp4n05</td>
<td>00:05:21</td>
</tr>
<tr>
<td>sp4n01.1940.0</td>
<td>markr</td>
<td>6/2 15:45</td>
<td>R</td>
<td>50</td>
<td>serial_l</td>
<td>sp4n05</td>
<td>00:05:22</td>
</tr>
<tr>
<td>sp4n01.1939.0</td>
<td>markr</td>
<td>6/2 15:45</td>
<td>R</td>
<td>50</td>
<td>serial_l</td>
<td>sp4n05</td>
<td>00:05:21</td>
</tr>
<tr>
<td>sp4n01.1938.0</td>
<td>markr</td>
<td>6/2 15:45</td>
<td>R</td>
<td>50</td>
<td>serial_l</td>
<td>sp4n05</td>
<td>00:05:22</td>
</tr>
</tbody>
</table>

9 job steps in queue, 1 waiting, 0 pending, 8 running, 0 held, 0 preempted

**Job resource data**

In this section, we describe the job resources used in accounting:

**machines**

To collect the job resource usage of machines consumed by a job. This is important if a job is vacated from one machine and restarted or started from a checkpoint on another machine. This is also important for parallel jobs if you want to observe resources consumed on various machines used for the parallel job.

**events**

Resource information based on an event or time can also be collected. You may wish to collect the accounting information every hour, day, week, or whenever. This is accomplished with the \texttt{llctl -g capture eventname} command.

Eventname is the identifier for the collected resource data.
Merging accounting files from nodes into one file

The global history file is created by merging all the local machine history files into one file. The default action is to collect the data from all machines listed in the administration file; alternatively, you could specify a list of machines:

```
llacctmrg [-h hosts] [-d directory]
```

The command writes job related data such as resource consumption to a file called globalhist.YYYYMMDDHHmm in GLOBAL_HISTORY or to the directory passed with the -d option:

```
[root@sp4n01]:/> llacctmrg -d /GPFS4/ACCT
llacctmrg: History transferred successfully from sp4n01.msc.itso.ibm.com (5756072 bytes).
llacctmrg: History transferred successfully from sp4n05.msc.itso.ibm.com (3094648 bytes).
```

On completion of the merge, the history files on the machines are reset to contain no data.

**Note:** If a LoadLeveler job issues a remote shell command to execute on another machine, then the resources used on the remote machine are not included in the accounting data.

9.1.1 Job account validation

In this section, we describe the account validation options in LoadLeveler.

**User configuration**

The job account numbers that you wish to allocate to a user is specified in the user stanza in the administration file:

```
usera:  type = user
        account = 1 4
```

When submitting a job, the user must add the following to his job command file:

```
# @ account_no = <accountno>
```

Interactive parallel jobs can export the LOADL_ACCOUNT_NO environment variable to achieve the same purpose as the account_no keyword in the job command file:

```
export LOADL_ACCOUNT_NO=1
```
System configuration
When account validation is required, the ACCT keyword must be set in the configuration file:

```
ACCT = ACCT_VALIDATE
```

LoadLeveler validates the job account number against known accounts specified in the user stanza. The validation of job account numbers is performed by the account validation program. The account validation program provided with the LoadLeveler package is `llacctval`, located in the release directory:

```
ACCT_VALIDATION = $(BIN)/llacctval
```

`llacctval` compares the job account number you specify in the job command file to the account numbers specified in the user stanza of the LoadL_admin file.

The `llsubmit` command invokes the account validation program specified in ACCT_VALIDATION and passes it the following parameters:

```
program user_name user_group user_acctn user_accts...
```

where:

- **program**: Name of the account validation program
- **user_name**: Name of the user to validate the account against
- **group_name**: The primary login group of `user_name`
- **user_acctn**: The account number stated in the job command file
- **user_accts**: A list of account numbers registered against
- **user_name**: In the user stanza from the administration file

If an account number match is found, then a zero value is returned; otherwise, a non-zero value is returned. On receipt of a non-zero value, `llsubmit` will reject the job for submission and return a non-zero value back to the shell. In the example output below, a job is submitted with the keyword account_number=100 in the command file. As the user is not registered to use this account, an error message is returned and the job fails to be submitted:

```
llsubmit: 2512-081 Account number "100" is not valid for user "markr".
llsubmit: 2512-051 This job has not been submitted to LoadLeveler.
```

As an example, we invoke the default `llacctval` program and pass it valid data. The return of the command is zero, which indicates that the account information matches:

```
[root@sp4n01]:/> /usr/lpp/LoadL/full/bin/llacctval markr staff 1 1
[root@sp4n01]:/> echo $?
0
```
We repeat the command again, but now we inform llacctval that the valid account for markr is 100. The program returns a non-zero value, indicating that the values do not match:

```
[root@sp4n01]:/> /usr/lpp/LoadL/full/bin/llacctval markr staff 1 100
[root@sp4n01]:/> echo $?
255
```

A simple example of an account validation program is given in Example 9-1. In order to utilize your own program, the ACCT_VALIDATION keyword must be modified to reference your validation program:

```
ACCT_VALIDATION = /u/loadl/bin/ouracctval.pl
```

**Example 9-1  Example account validation script**

```perl
#!/usr/bin/perl

@MYARRAY = @ARGV;

$value = 0;
$account = $MYARRAY[2];
@reg_accounts = splice @MYARRAY, 3;

undef @has_value;
for (@reg_accounts) { $has_value[$_] = 1; }

if ( $has_value[$account] == 0) {
  print "YOU ARE ATTEMPTING TO USE AN ACCOUNT FOR WHICH YOU ARE NOT REGISTERED\n"
  print "YOUR VALID ACCOUNTS ARE : @reg_accounts\n"
  $value = 255
}

exit $value
```

This script could be extended to mail a message informing the administrator of the attempted use, or even hook into the event management subsystem to inform operations staff of the failure.

**Note:** If you modify account information in the administration file and issue a reconfig, it will not impact jobs that are already running or queued.

### 9.1.2 API for custom reports

The GetHistory subroutine in the LoadLeveler API library libllapi.a is provided so you can write your own accounting reports. Example code can be found in the `/usr/lpp/LoadL/full/samples/llphist` directory that demonstrates how to interface to the subroutine.
You can use this API to generate your own reports. The `llhist` program in the sample directory reads in a global history file and outputs the data to a text file (Example 9-2):

```
llhist globalhistfile > outputfile
```

### Example 9-2  Data generated by `llhist` from a global history file

```
#Job Name:User:Group:Uid:Gid:Submit
Host:Steps:Requirements:Checkpoint?:Restart?:Job Type:Job Class:Wallclock
Limit:Procs:
Job Id:Image Size(Kb):Queued:Started:Finished:Nodes:User Time(s):System
Time(s):Total Time(s):Finish Status
```

```
ifs2:peterm:staff:8342:1:sp4n01.msc.itso.ibm.com:1:(Arch == "R6000") && (OpSys
== "AIX51"):no:yes:parallel:poe_l1:43200:2:
sp4n01.973.0:4:991067046:991067047:991067123:sp4n01.msc.itso.ibm.com,sp4n05.msc
```

If you do not want to collate an increasing amount of text files that you may need to later process with scripts, the data could be directly written into a database. The database approach has the advantage of allowing complex queries and trend analysis to be conducted easily.

### 9.1.3 Using CPU speed

A machine’s relative speed can be expressed as a value with the speed keyword in the node’s machine stanza in the LoadL_admin file. The higher the value, the more significant the weighting of the machine:

```
speed = <value>
```

You may want to have different charging fees for resource consumption on certain nodes. Where you have a mixture of node types, which all process the same job in different times, you may want to normalize the accounting so the charge is the same for all nodes with the keyword `cpu_speed_scale`:

```
cpu_speed_scale = true|false
```

In Table 9-1 on page 146, we show the impact on the reported `job_time` by the speed ratio.
### Table 9-1  Affects of cpu_speed on accounting job times

<table>
<thead>
<tr>
<th>Node</th>
<th>Job Time (seconds)</th>
<th>Speed Ratio</th>
<th>Job Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cpu_speed_scale</td>
<td></td>
<td>cpu_speed_scale</td>
</tr>
<tr>
<td></td>
<td>=false</td>
<td></td>
<td>= true</td>
</tr>
<tr>
<td>A</td>
<td>100</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>C</td>
<td>25</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>D</td>
<td>5</td>
<td>20</td>
<td>100</td>
</tr>
</tbody>
</table>

#### 9.1.4 Interaction with LL/SP/AIX accounting

Account reports can be generated with use of a job summary command: `llsummary`. If the command is executed without any arguments, it will use the local history files of the machine it was run on. Alternatively, existing global history files can also be used as input to generate a summary report by passing a file name.

<table>
<thead>
<tr>
<th>Name</th>
<th>Jobs</th>
<th>Steps</th>
<th>Job Cpu</th>
<th>Starter Cpu</th>
<th>Leverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>peterm</td>
<td>733</td>
<td>733</td>
<td>17:15:10</td>
<td>00:03:20</td>
<td>310.6</td>
</tr>
<tr>
<td>markr</td>
<td>33</td>
<td>33</td>
<td>02:40:32</td>
<td>00:00:10</td>
<td>963.2</td>
</tr>
<tr>
<td>kannan</td>
<td>95</td>
<td>95</td>
<td>00:00:23</td>
<td>00:00:23</td>
<td>1.0</td>
</tr>
<tr>
<td>TOTAL</td>
<td>861</td>
<td>861</td>
<td>19:56:06</td>
<td>00:03:54</td>
<td>306.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Jobs</th>
<th>Steps</th>
<th>Job Cpu</th>
<th>Starter Cpu</th>
<th>Leverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>poe_l1</td>
<td>718</td>
<td>718</td>
<td>07:58:36</td>
<td>00:03:21</td>
<td>142.9</td>
</tr>
<tr>
<td>poe_l2</td>
<td>1</td>
<td>1</td>
<td>02:05:54</td>
<td>00:00:00</td>
<td>(undefined)</td>
</tr>
<tr>
<td>serial_s</td>
<td>54</td>
<td>54</td>
<td>06:03:08</td>
<td>00:00:11</td>
<td>1980.7</td>
</tr>
<tr>
<td>serial_l</td>
<td>67</td>
<td>67</td>
<td>03:48:22</td>
<td>00:00:16</td>
<td>856.4</td>
</tr>
<tr>
<td>poe_l</td>
<td>5</td>
<td>5</td>
<td>00:00:00</td>
<td>00:00:00</td>
<td>(undefined)</td>
</tr>
<tr>
<td>inter_class</td>
<td>14</td>
<td>14</td>
<td>00:00:04</td>
<td>00:00:03</td>
<td>1.3</td>
</tr>
<tr>
<td>express</td>
<td>1</td>
<td>1</td>
<td>00:00:00</td>
<td>00:00:00</td>
<td>(undefined)</td>
</tr>
<tr>
<td>serial</td>
<td>1</td>
<td>1</td>
<td>00:00:00</td>
<td>00:00:00</td>
<td>(undefined)</td>
</tr>
<tr>
<td>TOTAL</td>
<td>861</td>
<td>861</td>
<td>19:56:06</td>
<td>00:03:54</td>
<td>306.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group</th>
<th>Jobs</th>
<th>Steps</th>
<th>Job Cpu</th>
<th>Starter Cpu</th>
<th>Leverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>No_Group</td>
<td>844</td>
<td>844</td>
<td>18:11:29</td>
<td>00:03:46</td>
<td>289.8</td>
</tr>
<tr>
<td>AWE</td>
<td>17</td>
<td>17</td>
<td>01:44:37</td>
<td>00:00:07</td>
<td>896.7</td>
</tr>
<tr>
<td>TOTAL</td>
<td>861</td>
<td>861</td>
<td>19:56:06</td>
<td>00:03:54</td>
<td>306.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Account</th>
<th>Jobs</th>
<th>Steps</th>
<th>Job Cpu</th>
<th>Starter Cpu</th>
<th>Leverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>847</td>
<td>847</td>
<td>18:20:02</td>
<td>00:03:47</td>
<td>290.8</td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>14</td>
<td>01:36:04</td>
<td>00:00:06</td>
<td>960.7</td>
</tr>
<tr>
<td>TOTAL</td>
<td>861</td>
<td>861</td>
<td>19:56:06</td>
<td>00:03:54</td>
<td>306.7</td>
</tr>
</tbody>
</table>

---

**Node Job Time (seconds) cpu_speed_scale=false**

- A: 100
- B: 50
- C: 25
- D: 5

**Speed Ratio**

- A: 1
- B: 2
- C: 4
- D: 20

**Node Job Time (seconds) cpu_speed_scale=true**

- A: 100
- B: 100
- C: 100
- D: 100
Above is the default report summary; `llsummary` can also generate other formats, including throughput. For a full listing of the possible options, consult the manual *LoadLeveler V2R2 for AIX: Using and Administering*, SA22-7311.

**SP exclusive accounting**

The standard AIX accounting facility produces accounting records for each process that completes execution. Accounting records are then merged by the user. Unfortunately, this results in the loss of processes that were exclusively using that node.

LoadLeveler differentiates a user job for exclusive accounting by generating process records. LoadLeveler runs a start job command as the user of the job when a job begins. Once the job exits, an end job command is also run as the user. As these commands generate process accounting entries for the user, it allows the job information to be separated from the other processes on the node. The start and end commands of the job are used to calculate the exclusive time of the user job used on the node.

This SP exclusive accounting facility for a job step is enabled with the use of the following keyword:

```
spacct_excluse_enable = true | false
```

In order for this keyword to be effective exclusive, user accounting must be enabled in the SP object of the SDR. The node level SDR attribute `acct_excluse_enable` specifies whatever accounting is activated to process start and end job records for exclusive jobs.

### 9.2 Monitoring

In this section, we describe some examples of monitoring the job execution in the LoadLeveler environment.

#### 9.2.1 Customizing the LoadLeveler GUI

One of the monitoring options is to customize the LoadLeveler GUI by using the resource file.
The xloadl resource file
You can make changes to the look and functionality of the xloadl GUI using the .Xdefaults file. By default, the resources file for xloadl is located in /usr/lpp/LoadL/full/lib. In order to make modifications locally, your .Xdefaults file should contain the resource variables you wish to modify. The structure of the xloadl resource file is documented in Loadleveler V2R2 for AIX: Using and Administering, SA22-7311.

Modifying the look and feel
The general look and feel of xloadl, such as the foreground and background colors, fields being displayed, field lengths, and fonts can be modified by the user. Below, we redefine the displayable field lengths and their labels for the job window and store them in our .Xdefaults file. A length of zero for a data field prevents the field from being displayed:

*jobid_len: 15
*jobid_label: Id | Name
*job_owner_len: 5
*job_owner_label: Owner
*job_dh_len: 0
*job_dh_label: Hold Until
*job_stat_len: 2
*job_stat_label: ST
*job_prio_len: 0
*job_prio_label: PRI
*job_class_len: 7
*job_class_label: Class
*job_jt_len: 7
*job_jt_label: Type
*job_host_len: 6
*job_host_label: Run On
*job_nh_len: 2
*job_nh_label: #H

Creating your own pulldown menus
Creating your own pulldown menus is possible using the provided userJobPulldown and userMachinePulldown resources variables. The structure is given below, with Example 9-3 on page 149 to demonstrate the modifications. The sample output is shown in Figure 9-1 on page 150.

user<label>_Option# True activates the pulldown; false to deactivate
user<label>_command The command you wish to run
user<label>_labelString Title of the pulldown
user<label>_parameter True to send selected job or machines names to the command, false to ignore selections
user<label>_output  Window | stdout | filename | none

Where:

► window: Output goes to an Information window
► filename: Output goes to a file (Absolute path)
► stdout: Output goes to the messages window
► none: No output

Note: If your path is not set correctly, then xloadl will not be able to find any executables you call from the resource variables.

Example 9-3  User defined pulldowns

*userJobPullDown : True
*userJob.labelString : Options
*userJob_Option1: True
*userJob_Option1_command: llq -f %o %a %dd %dq %jn %jt %c %nh %st
*userJob_Option1.labelString: llq(messages)
*userJob_Option1_parameter: false
*userJob_Option1_output: stdout

*userJob_Option2: True
*userJob_Option2_command: llq -f %o %a %dd %dq %jn %jt %c %nh %st
*userJob_Option2.labelString: llq(window)
*userJob_Option2_parameter: false
*userJob_Option2_output: window

*userMachine_Option1 :True
*userMachine_Option1_command: class.pl
*userMachine_Option1.labelString: Classes(messages)
*userMachine_Option1_parameter: True
*userMachine_Option1_output: stdout
**Figure 9-1** xloadl showing customized output
9.2.2 Customizing llq displays

Sometimes it is useful to output job queue information in a form that can be easily parsed by scripts. This can be achieved with the -r flag, which outputs raw information delimited by a ! character. In the example below, we request job information in this order: dispatch date, queue date, class, node, type, user, priority, and status:

```
llq -r %dd %dq %c %jn %jt %nh %o %p %st
```

```
105/28/2001 15:54!serial_l!sp4n01.msc.itso.ibm.com.1080!Serial!0!peterm!100!I
```

If you want your llq output in a custom format, but do not want to have to remember the various flags, create a small llq wrapper script similar to Example 9-4.

Example 9-4 Example of a llq wrapper using customized options

```
#!/bin/ksh
/usr/lpp/LoadL/full/bin/llq -f %o %a %dd $dh %dq %jn %jt %c %nh %st
```

Essentially, the wrapper calls the llq command complete with flags. Place the script in your home directory and modify your path so that your local llq is called before the llq binary in the LoadLeveler release directory. In the llq wrapper, we have used the -f flag to request the output formatted complete with column headers:

```
Owner    Account Disp. Date Queue Date Job Name  Type Class    NM     ST
-------- -------- ----------- ----------- ----------------- ---- ----------- --
```

```
kannan           06/02 21:14 sp6en0.msc.itso.ibm. PAR  express    1      H
kannan           06/02 21:14 sp6en0.msc.itso.ibm. PAR  express    1      H
kannan           06/02 21:14 sp6en0.msc.itso.ibm. PAR  express    1      H
```

**Note:** The -r and -f format flags cannot be used in conjunction with the -x or -l flags.
9.2.3 Other monitoring tools

Several sophisticated applications have been written to display the status of LoadLeveler clusters. This section contains several figures that will illustrate what can be achieved using the APIs.

NERD

NERD is the Node Environment Resource Display application, written by Phil Renwick of AWE Aldermaston in the UK. NERD is a C++ application that extracts SP and LoadLeveler information and builds a Web page displaying the status of the SP and LoadLeveler jobs. It runs periodically as a cron job and users view the Web page to see how busy the machine is. The output of this application is shown in Figure 9-2 on page 153.
Figure 9-2  NERD application
LoadView

LoadView was written by Dave Michaud of the National Centers for Environment Prediction (NCEP) in the USA. LoadView uses a number of scripts to extract information from the SP SDR and LoadLeveler periodically. The Graphical interface is a Java Applet, which displays the extracted data, automatically updating the information. Figure 9-3 shows the output of this application.

![LoadView application](image)

Figure 9-3   LoadView application
LLAMA
LLama is the LoadLeveler Advanced Monitoring Application, written by Andy Pierce of IBM. The output of LLAMA is shown in Figure 9-4.

9.3 Mail notification

By default, LoadLeveler sends e-mail to the user specified in the job command file when a job has completed. The user should specify:

```bash
# @ notification = always | error | never | start | complete
# @ notify_user = fred@bloggs.com
```
Typical e-mails from a starting job, a failed job, a cancelled job, and a completed job are shown in the following four examples (Example 9-5 to Example 9-8 on page 157).

Example 9-5  E-mail sent by a job starting
From: LoadLeveler

LoadLeveler Job Step: step1
Your job step, "node09.stc.uk.ibm.com.471.0" has started.

Token status:

Starter Information:
    Submitted: Tue Jun  5 20:02:53 2001
    Executable: single.cmd
    Job Step Type: Parallel
    Number of Tasks: 2
    Host Name: 
------------------

Example 9-6  E-mail sent by a failing job
From: LoadLeveler

LoadLeveler Job Step: node09.stc.uk.ibm.com.471.0
    Executable: single.cmd
    Executable arguments:
    State for machine: node13.stc.uk.ibm.com
        LoadL_starter: Hard WALL CLOCK limit exceeded. Soft limit -1, Hard limit 30
        State for machine: node14.stc.uk.ibm.com
        LoadL_starter: Received remove order, step will be terminated.

This job step was dispatched to run 1 time(s).
This job step was rejected by Starter 0 time(s).
Submitted at: Tue Jun  5 20:02:53 2001
Started at: Tue Jun  5 20:02:54 2001
The job step was canceled by at: Tue Jun  5 20:03:24 2001
Exited at: Tue Jun  5 20:03:24 2001
    Real Time:  0 00:00:31
    Job Step User Time:  0 00:00:07
    Job Step System Time:  0 00:00:01
    Total Job Step Time:  0 00:00:08
Example 9-7  E-mail sent by a cancelled job
From: LoadLeveler

LoadLeveler Job Step: node09.stc.uk.ibm.com.472.0
Executable: single.cmd
Executable arguments:
State for machine: node14.stc.uk.ibm.com
LoadL_starter: Received remove order, step will be terminated.

State for machine: node15.stc.uk.ibm.com
LoadL_starter: Received remove order, step will be terminated.

This job step was dispatched to run 1 time(s).
This job step was rejected by Starter 0 time(s).
Submitted at: Tue Jun  5 20:05:06 2001
Started at: Tue Jun  5 20:05:07 2001
The job step was canceled by peterm at: Tue Jun  5 20:05:25 2001
Exited at: Tue Jun  5 20:05:25 2001

Real Time:   0 00:00:19
Job Step User Time:   0 00:00:12
Job Step System Time:   0 00:00:01
Total Job Step Time:   0 00:00:13

Starter User Time:   0 00:00:00
Starter System Time:   0 00:00:00
Total Starter Time:   0 00:00:00

Example 9-8  E-mail sent by a completed job
From: LoadLeveler

LoadLeveler Job Step: node09.stc.uk.ibm.com.473.0
Executable: single.cmd
Executable arguments:
State for machine: node16.stc.uk.ibm.com
LoadL_starter: The program, single.cmd, exited normally and returned an
   exit code of 0.

State for machine: node13.stc.uk.ibm.com
Within the LoadL_config file, the administrator can specify an alternate mail notification program:

MAIL = /u/loadl/LoadL_mail

This program is passed three arguments, exactly as would be passed to the standard mail program:

/u/loadl/LoadL_mail -s <job_name> <notify_user>

It receives the body of the mail message to standard input.

By specifying an alternate mail program and parsing the input, administrators may customize the notification method.

When the job starts and the user has specified notification = start, the mail program runs on the machine running the LoadL_starter daemon and is run with the UID/GID of the job owner. However, the mail program runs when the job completes and the user has specified notification = complete. This runs on the machine that scheduled the job with the UID/GID of the LoadLeveler administrator, after the LoadL_starter daemon has indicated that this job has completed. This is precisely what is needed to extract the accounting records for the job, which are written on the schedd machine when the job finishes, and are accessible only to the LoadLeveler administrator.

The Perl script in Example 9-9 was used as an alternative mail program, and instead of mailing the summary shown above, mails the full job statistics back to the submitting user.

Example 9-9 Sample alternate mail notification program

```perl
#!/usr/bin/perl
#
# Alternate mail notification program
```

---
# Read everything in from stdin
# @EVERYTHING = <STDIN>;
#
# Extract the name of the job
#
foreach $_ (@EVERYTHING) {
    chop;
    if (/LoadLeveler Job Step:/) {

    # When you have found it, get the accounting information
    ($junk,$junk,$junk,$jobname) = split;
    @pieces = split(/\./, $jobname);
    $n = @pieces;
    # Discard the step number to get the job name
    splice(@pieces,$n-1);
    $newjobname = join('.',@pieces);
    # Get the full accounting information for the job
    @answer = `llsummary -l -j $newjobname 2>&1`;
    }
}
#
# Mail the accounting information back
# to the user
#
open (MAILOUT, "| mail @ARGV");
foreach $line (@answer) {
    print MAILOUT "$line\n";
}
close(MAILOUT);

An alternative use of the mail notification program might be to alert the user when a job has completed by popping up a window on the submitting workstation. Suppose the user sets the DISPLAY environment variable in the submitting interactive session, and sets @ environment = $DISPLAY in the job command file. Then the output from the llsummary -l command includes a line of the form:
Env: DISPLAY=a.b.c.d:0;
which can be located and used in the mail notification program. For example, if the last few lines in Example 9-9 on page 158 are replaced with the code in Example 9-10 and Example 9-11, then a window will be opened on the submitting workstation.

Example 9-10  Modified tail of LoadL_mail

```bash
system "at now << EOF
/u/loadl1/LoadL_mail.part2 $newjobname
EOF"
```

Example 9-11  LoadL_mail.part2 perl script

```perl
#!/usr/bin/perl
#
# Alternate mail notification program
#
$newjobname = $ARGV[0];
#
# Get the full accounting information for the job
#
@answer = `llsummary -l -j $newjobname 2>&1`;
#
# Extract the DISPLAY if it has been passed by the job
#
foreach $_ (@answer) {
    chop;
    if (/ Env:/) {
        ($junk, @therest) = split (/:/,$_);
        $env = join (":", @therest); # stick line back together
    }
} @values = split (/

```

Clearly, more sophisticated uses are possible. The second example assumes that the user has remembered to export the DISPLAY variable, and pass it through to the job using the `# @` environment directive. A more general solution would be to use the Zephyr messaging system to alert the user.
9.4 Managing log files

To aid in the diagnosis of problems encountered in the cluster, it is necessary to maintain good housekeeping. Of course, administrators may decide to turn off logging until a problem arises, but this will not help if you are requested to send information to a support team for a problem that you can not reproduce.

9.4.1 What are the log files and locations

The directory location of the log files is defined in the LoadL_config file:

\[
\text{LOG} = \$(\text{tilde})/\text{log}\n\]

We recommend creating a separate logical volume other than /var to store the log files, such as /var/loadl, in order to prevent issues related to filling the var file system. In the same way, you can also create a separate file system for spool and executable directories.

Each of the daemons have a log keyword that allows you to specify the location and name of each daemon log file:

\[
\text{STARTD\_LOG} = \$(\text{LOG})/\text{StartLog}\n\]

Each of these files can be instructed to limit their size to a set maximum in byte units of which the default is 64Kb:

\[
\text{MAX\_STARTD\_LOG} = 1000000\n\]

The time taken for this log to reach its preset level is dependent on the amount of traffic encountered and the debug level set. An increased debug level will obviously lead to an increase in data being logged and, hence, the log files being rotated more frequently.

LoadLeveler also creates temporary log files that the starter daemon uses. When a job starts, a StarterLog.pid file is created and is present for the duration of the job. Once the job completes, that log information is appended to the StarterLog file.

9.4.2 How to save log files

When a log file reaches its preset maximum limit, it is renamed as filename.old and a new daemon log file is created. The following example shows old and new log files:

- rw-r--r--  1 loadl loadl          4581 May 28 13:55 NegotiatorLog
- rw-r--r--  1 loadl loadl         64074 May 28 13:49 NegotiatorLog.old
Unfortunately, this means that only two versions of the log file can exist and these can easily be overwritten on a busy system, preventing historical problem determination at a later point in time.

With the SAVELOGS = <directory> keyword, rotated log files are uniquely named in the form <daemon name, time, machine name> and written to the specified directory. For example:

```
NegotiatorLogMay16.08:00:00sp4n01.itso.msc.ibm.com
```

It is your responsibility to manage the logs and the necessary space required. We recommend careful thought before referencing non-local directories for log files, due to the possibility of an inaccessible directory.

We recommend that, unless space is an issue, store the logs locally first. Then, via an automated script, either transfer them to a global file system or migrate the logs to a tape subsystem, such as Tivoli Storage Manager.

### 9.4.3 Adding debug output to the log files

Additional debugging can be produced by the LoadLeveler daemons by specifying debug flags in the LoadL_config file. Initially, the following keywords are not set:

- /SM590000 KBDD_DEBUG
- /SM590000 STARTD_DEBUG
- /SM590000 SCHEDD_DEBUG
- /SM590000 NEGOTIATOR_DEBUG
- /SM590000 GSMONITOR_DEBUG
- /SM590000 STARTER_DEBUG
- /SM590000 MASTER_DEBUG

This produces a default level of logging, as described above. Any of the flags described in Table 9-2 may be set to log specific types of daemon activity. YES means that adding the flag produces additional output for the particular daemon, and NO means that the flag does not affect the output produced by the daemon.

#### Table 9-2 The effect of adding debug flags

<table>
<thead>
<tr>
<th>Debug flag</th>
<th>Master</th>
<th>Negotiator</th>
<th>Schedd</th>
<th>Startd</th>
<th>Starter</th>
<th>Kbdd</th>
</tr>
</thead>
<tbody>
<tr>
<td>D_ACCOUNT</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>D_AFS</td>
<td></td>
<td>(Not Tested)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D_DAEMON</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
A full description of the different debug flags is given in *Loadleveler V2R2 for AIX: Using and Administering*, SA22-7311.

Several points should be noted here:

- Several of the flags are effectively defined to be the same. These are the daemon-specific flags D_NEGOTIATE, D_SCHEDD, D_STARTD, and D_STARTER. It is assumed that the relevant flag will only be set against its own daemon. Setting STARTER_DEBUG = D_NEGOTIATE works, as D_NEGOTIATE and D_STARTER map to the same value.

- D_FULLDEBUG generates a lot of output, but more can be produced by setting the flags individually.
D_LOCKING produces a very large amount of output!

9.4.4 How to track a job through the system

The process of determining why a job failed is a common task for administrators. It is necessary to know which machines run which daemons and which will point to the machine containing the necessary information in the logs.

One method would be to use a script to retrieve the job information from the scheduling machine, then connect to the machines that ran the job. The information from the machines would be parsed from the necessary logs, processed, and the output (Example 9-12) condensed.

Example 9-12 Output of script to search for a job on allocated machines

-----------> Allocated Nodes for sp4n01.msc.itso.ibm.com.550.0
sp4n01.msc.itso.ibm.com.

-----------> /u/loadl/log/StarterLog
sp4n01.msc.itso.ibm.com.

05/16 11:00:12 TI-0 <uninitialized Job> Sending request for job step, sp4n01.msc.itso.ibm.com.550.0
05/16 11:00:12 TI-0 sp4n01.550.0 Sending request for job step, sp4n01.msc.itso.ibm.com.550.0
05/16 11:00:12 TI-0 sp4n01.550.0 Sending request for job step, sp4n01.msc.itso.ibm.com.550.0
05/16 11:00:12 TI-0 sp4n01.550.0 Prolog not run, no program was specified.
05/16 11:00:12 TI-0 sp4n01.550.0 run_dir = /u/loadl/execute/sp4n01.msc.itso.ibm.com.550.0
05/16 11:00:12 TI-0 sp4n01.550.0 run_dir = /u/loadl/execute/sp4n01.msc.itso.ibm.com.550.0
05/16 11:00:12 TI-0 sp4n01.550.0 Sending request for executable to Schedd
05/16 11:00:13 TI-0 sp4n01.msc.itso.ibm.com.550.0 Sending request for credentials to Schedd
05/16 11:00:13 TI-0 05/16 11:00:13 TI-0 sp4n01.550.0 User environment prolog not run, no program was specified.
05/16 11:00:13 TI-0 sp4n01.550.0 llcheckpriv program exited, termsig = 0, coredump = 0, retcode = 0
05/16 11:00:13 TI-0 sp4n01.550.0 Sending READY status to Startd
05/16 11:00:13 TI-0 sp4n01.550.0 Main task program started (pid=24832 process count=1).
05/16 11:00:13 TI-0 sp4n01.550.0 Sending SOME_RUNNING status to Startd
05/16 11:00:13 TI-0 LoadLeveler: sp4n01.550.0 CPU soft limit (9223372036854775807 seconds) for job step being adjusted down to the hard limit (2147483647 seconds).
05/16 11:00:13 TI-0 sp4n01.550.0 Sending PassOpenSocket command.
05/16 11:00:13 TI-0 sp4n01.550.0 All parallel task started, command = /etc/pmdv3 (pid=25792 process count=2).
05/16 11:00:13 TI-0 sp4n01.550.0 Sending RUNNING status to Startd
05/16 11:00:13 TI-0 LoadLeveler: sp4n01.550.0 CPU soft limit (9223372036854775807 seconds) for job step being adjusted down to the hard limit (2147483647 seconds).
05/16 11:42:39 TI-0 sp4n01.550.0 Task exited, termsig = 0, coredump = 0, retcode = 161
05/16 11:42:39 TI-0 sp4n01.550.0 Sending signal 15 to task 0 process (25792).
05/16 11:42:44 TI-0 sp4n01.550.0 Task exited, termsig = 0, coredump = 0, retcode = 0
05/16 11:42:44 TI-0 sp4n01.550.0 User environment epilog not run, no program was specified.
9.4.5 Other relevant log files

In this section, we describe the other log files that you should manage in a LoadLeveler environment.

Error log

The AIX error log contains many messages written by the errloger. The `errpt` command can be used to display these messages:

```
errpt | more
```

<table>
<thead>
<tr>
<th>IDENTIFIER</th>
<th>TIMESTAMP</th>
<th>T</th>
<th>C</th>
<th>RESOURCE_NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>9A8FB4A9</td>
<td>0528131901</td>
<td>U</td>
<td>S</td>
<td>LoadLeveler</td>
<td>SOFTWARE</td>
</tr>
<tr>
<td>9A8FB4A9</td>
<td>0528131901</td>
<td>U</td>
<td>S</td>
<td>LoadLeveler</td>
<td>SOFTWARE</td>
</tr>
<tr>
<td>9A8FB4A9</td>
<td>0528131901</td>
<td>U</td>
<td>S</td>
<td>LoadLeveler</td>
<td>SOFTWARE</td>
</tr>
<tr>
<td>9A8FB4A9</td>
<td>0528131901</td>
<td>U</td>
<td>S</td>
<td>LoadLeveler</td>
<td>SOFTWARE</td>
</tr>
<tr>
<td>12081DC6</td>
<td>0528131901</td>
<td>P</td>
<td>S</td>
<td>harmld</td>
<td>SOFTWARE PROGRAM ERROR</td>
</tr>
<tr>
<td>9A8FB4A9</td>
<td>0528131801</td>
<td>U</td>
<td>S</td>
<td>LoadLeveler</td>
<td>SOFTWARE</td>
</tr>
<tr>
<td>9A8FB4A9</td>
<td>0528131801</td>
<td>U</td>
<td>S</td>
<td>LoadLeveler</td>
<td>SOFTWARE</td>
</tr>
<tr>
<td>A67EF6DC</td>
<td>0528131801</td>
<td>P</td>
<td>S</td>
<td>LoadLeveler</td>
<td>SOFTWARE PROGRAM ABNORMALLY TERMINATED</td>
</tr>
</tbody>
</table>

Analyzing this data can be useful when determining problems, such as a job failure due to a user application crash or if another administrator had issued a command, such as the LoadLeveler reconfiguration request below:

```
--------------------------
LABEL:          LL_INFO_ST
IDENTIFIER:     9A8FB4A9
Date/Time:       Mon May 28 10:37:33 EDT
Sequence Number: 27077
Machine Id:      006030174C00
Node Id:         sp4n01
Class:           S
Type:            UNKN
Resource Name:   LoadLeveler
Description     SOFTWARE

Probable Causes
APPLICATION PROGRAM
```
User Causes
OPERATOR
USER GENERATED SIGNAL

Recommended Actions
NO ACTION NECESSARY

Detail Data
DETECTING MODULE
LPP=LoadL,Fn=MasterCommands.C,SID=1.27,L#=299,
DIAGNOSTIC EXPLANATION
Got RECONFIG command

This data can be extracted and provide helpful information determining events and commands to extract LoadLeveler entries from the AIX error log:

```
[root@sp4n01]:/> errpt -N LoadLeveler -d S -T PERM
IDENTIFIER   TIMESTAMP   T C RESOURCE_NAME   DESCRIPTION
A67EF6DC   0528131801 P S LoadLeveler    SOFTWARE PROGRAM ABNORMALLY TERMINATED
A67EF6DC   0527172901 P S LoadLeveler    SOFTWARE PROGRAM ABNORMALLY TERMINATED
A67EF6DC   0527163201 P S LoadLeveler    SOFTWARE PROGRAM ABNORMALLY TERMINATED
A67EF6DC   0522151801 P S LoadLeveler    SOFTWARE PROGRAM ABNORMALLY TERMINATED
```

Syslog
The syslog daemon receives messages sent to it according to the documented syslog API. These received messages are either ignored or written to log files according to the rules in the syslog configuration file /etc/syslog.conf:

```
# "mail messages, at debug or higher, go to Log file. File must exist."
# "all facilities, at debug and higher, go to console"
# "all facilities, at crit or higher, go to all users"
# mail.debug /usr/spool/mqueue/syslog
# *.debug /dev/console
# *.crit *
# *.debug /tmp/syslog.out rotate size 100k files 4
# *.crit /tmp/syslog.out rotate time 1d
daemon.notice /var/adm/SPlogs/SPdaemon.log
```

Below is an example of the type of LoadLeveler messages that can occur in the syslog (in this case, to /var/adm/SPlogs/SPdaemon.log):

```
May 28 10:48:10 sp4n01 LoadLeveler[38290]: LPP=LoadL,Fn=MasterCommands.C,SID=1.27,L#=299, Got RECONFIG command

May 28 12:16:38 sp4n01 LoadLeveler[47072]: LPP=LoadL,Fn=Master.C,SID=1.69.1.14,L#=1468,
                        Started LoadL_startd, pid and pgroup = 46420
```
9.5 Troubleshooting

The process of determining why a job failed is a common task for administrators. It is necessary to know which machines run which daemons and which will point to the machine that contains the necessary information in the log files.

9.5.1 Managing Loadleveler files

It is common for an administrator to want to test a new configuration. You could make copies of the existing config or, alternately, you use the /etc/LoadL.cfg to point to an alternative configuration.

The configurations used during this redbook were kept in CONFIG/TEST<x> in the LoadLeveler user home directory. To change the configuration, the cluster was stopped globally:

```
/usr/lpp/LoadL/full/bin/llctl -g stop
```

The file /etc/LoadL.cfg is modified to point to the new LoadLConfig, as shown in Example 9-13.

**Example 9-13 /etc/LoadL.cfg - demonstrating an alternative cluster configuration**

```
LoadL_Userid = loadl
LoadL_Group = loadl
LoadLConfig = /u/loadl/CONFIG/TEST4/LoadL_config
```

The /etc/LoadL.cfg file is then copied across the nodes in the cluster and then the cluster is started:

```
pcp -av /etc/LoadL.cfg /etc/LoadL.cfg
/usr/lpp/LoadL/full/bin/llctl -g start
```

The necessary test cases should be run against the new config to verify that the configuration is operating as expected. Once the test is complete, you can either maintain the configuration, if successful, or switch back to the previous config, if further work is required.
9.5.2 Schedd node failure

When a scheduler node fails, the resources that were allocated to jobs scheduled by the node are unavailable until the daemon is restarted. It is possible to recover the resources using the `llct1 purgeschedd` command to remove all the jobs scheduled by the down scheduler node. This is only applicable when used in conjunction with the schedd_fenced keyword instructing the Central Manager to ignore the failed nodes. The following procedure can be used to recover from the schedd node failure:

1. You identify that a scheduler node has failed and will be down for a significant amount of time.
2. The schedd_fenced=true keyword is added to the failed machine’s stanza in the administration file.
3. Run `llct1 reconfig` on the Central Manager.
4. Run `llct1 -h node purgeschedd`, which instructs the node to purge all the scheduled jobs.
5. On the failed node, remove all files from the LoadLeveler spool directory.
6. When the failed node returns to service then remove the schedd_fenced=true from the administration file.
7. Run `llct1 reconfig` on the Central Manager.

9.5.3 Job not running

The `llq -s` command can be used to determining why a job remains in the idle queue instead of running.

Note that the `llq -s` option may not be able to tell why a job is not running in every possible situation. When it cannot give enough information, the log files need to be examined.
9.5.4 Job command files

The following describe possible problems with the job command files:

- Ensure that your job command files are readable.
- If you are using the job command files produced by other users, they may need editing.
- Ensure that standard output and standard error file names are valid.
During this project, we have collected several tools. This appendix provides the source code for procedures used in this book. You may wish to use these tools and customize them to suit your requirement.
Example A-1 is a sample program for the tool llbeans.

Example: A-1  llbeans.c

/***************************************************************************/
llbeans uses the LL data access api to gain information about nodes and
jobs. format is:
   llbeans -r -w -v
    -r print running jobs, -w prints waiting jobs, -v prints
     a job jist for running jobs.
7/20/00 - JS, initial version for NAVO
***************************************************************************/

#include "/usr/lpp/LoadL/full/include/llapi.h"/* Get lib calls, data types and
structures */
#include <time.h>
#include <string.h>
#define MATCH 0
#define B_SIZE 1000
#define HEXDOT 46
#define ALL 0
#define TRUE 1
#define FALSE 0
#define WAITING 1
#define RUNNING 2
int report(int rc, char *Message) { /* subroutine to exit on error */
   printf("%s. rc = %d\n",Message,rc);
   exit (rc);
} /* end of report subroutine */

int printit(int state, int OutPrint, char *OutBuff) {
   if (state == STATE_RUNNING)  {  /* this is a running job   */
      if (OutPrint==RUNNING || OutPrint==ALL)  { /* user wants to see running
jobs */
         printf("%s\n",OutBuff); /*  Print first line of job output */
      } else {  /* job not running */
         if (OutPrint==WAITING || OutPrint==ALL) { /* user wants to see waiting
jobs */
            printf("%s\n",OutBuff); /* Print first line of job output */
    
};
/*  end of printit subroutine */

int main(int argc, char *argv[])
{
  /* Define local variables */
  LL_element *queryObject=NULL, *job=NULL, *jobObject=NULL;
  int i,j,rc, num, err, state, numnodes, numtasks, WCLimit, Junk, c;
  int OutPrint=ALL, PrintRun=FALSE, PrintWait=FALSE, Verbose=FALSE;
  int minnodes, numinit; /* min node count and number of initiators */
  LL_element *step=NULL, *machine = NULL, *cred=NULL, *node=NULL;
  char *id=NULL, *name=NULL, *JobClass=NULL, *Comment=NULL, *JobName=NULL;
  time_t endtime, SubmitTime;
      *TempBuff;
  char *Message;
  size_t smax=B_SIZE;
  struct tm  *LocalTime;
  TextTime = (char *)malloc(B_SIZE);
  LocalTime = (struct tm *)malloc(B_SIZE);
  NewBuff = (char *)malloc(B_SIZE);
  LLID = (char *)malloc(B_SIZE);
  LLStep = (char *)malloc(B_SIZE);
  LocalMach = (char *)malloc(B_SIZE);
  TempBuff = (char *)malloc( B_SIZE);
  OutBuff = (char *)malloc(5 * B_SIZE);
  /***********id = (char *)malloc(B_SIZE);***********/
  name = (char *)malloc(B_SIZE);
  JobClass = (char *)malloc(B_SIZE);
  Message = (char *)malloc(B_SIZE);

  /* Note: option code from p117 K&R */
  while ( --argc > 0 && (*++argv)[0] == '-' )  { /* for all the input
       arguments... */
    while (c = **argv[0]) /* note that this reads each option character */
      switch (c) {
        case 'r':
          PrintRun = TRUE;
          break;
        case 'w':
        case 't':
          PrintWait = TRUE;
          break;
        case 'v':
          Verbose = TRUE;
          break;
        case 'j':
          junk = strncat(junk, **argv[0], junksize - junk);
          break;
      }
PrintWait = TRUE;
break;
case 'v':
    Verbose = TRUE;
    break;
default:
    printf("llbeans format is 'llbeans -r -w -v\n'\n");
    printf(" where: -r prints all running jobs\n");
    printf(" -w prints all waiting jobs\n");
    printf(" -v prints a node list for running jobs\n");
    exit(-1);
    break;
}
/* end of all input args */
/* figure out what to print */
if (PrintRun && !PrintWait) { /* if only PrintRun set */
    OutPrint = RUNNING; /* only print running */
}
if (!PrintRun && PrintWait) { /* If only printWait set */
    OutPrint = WAITING; /* only print waiting */
}

/* Initialize the query for jobs */
queryObject = ll_query(JOBS);

/* I want to query all jobs */
rc = ll_set_request(queryObject,QUERY_ALL,NULL,NULL);
if (rc != 0) { sprintf(Message, "ll_set_request failed"); report(rc, Message); }

/* Request the objects from the Negotiator daemon */
job = ll_get_objs(queryObject,LL_CM,NULL,&num,&err);

/* Did we get a list of objects ? */
if (job == NULL) {
    printf(" There are no jobs in the job queue.\n");
} else {
    /* Loop through the list and process */
    while(job) {
        rc = ll_get_data(job,LL_JobSubmitTime, &SubmitTime); /* get the time 
job submitted */
        if (rc != 0) {printf(Message, "LL_JobSubmitTime failed"); report(rc, Message); }
        rc = ll_get_data(job,LL_JobName, &JobName);
        if (rc != 0) {printf(Message, "LL_JobGetJobName failed"); report(rc, Message); }
        rc = ll_get_data(job,LL_JobGetFirstStep, &step);
        if (rc != 0) {printf(Message, "LL_JobGetFirstStep failed"); report(rc, Message); }
while (step) {
    rc = ll_get_data(step, LL_StepState, &state);
    if (rc != 0) {sprintf(Message, "LL_StepState failed"); report(rc, Message); }
    for (i=0 ; i<B_SIZE ; ++i) { /* Clear 'dem buffers */
        *(TempBuff+i) = *(LocalMach+i) = *(LLStep+i) = *(NewBuff+i) = *(LLID+i) = 0;
    }
    for (i=0 ; i<(B_SIZE*5) ; ++i) {
        *(OutBuff+i) = 0;
    }
    rc = ll_get_data(step, LL_StepID, &id); /* get job ID */
    if (rc != 0) {sprintf(Message, "LL_StepID failed"); report(rc, Message); }
    TempPtr = strchr(id, HEXDOT); /* Format to use only local machine ID */
    Junk = TempPtr-id;
    strncpy(LocalMach,id,Junk);
    TempPtr = strrchr(id, HEXDOT);
    strcpy(LLStep, TempPtr);
    strncpy(NewBuff,id,(TempPtr-id)); /* hack off step ID */
    TempPtr = strrchr(NewBuff, HEXDOT);
    strcpy(LLID, TempPtr); /* save LL id value */
    strcat(LocalMach, LLID); /* these 2 steps add in LL job and step id's */
    strcat(LocalMach, LLStep);
    sprintf(OutBuff, "Job: %s %s %d ", LocalMach, JobName, state); /* save in output buffer */
    free(id);
    rc = ll_get_data(step, LL_StepDispatchTime, &endtime); /* get dispatch time */
    if (rc != 0) {sprintf(Message, "LL_StepDispatchTime failed"); report(rc, Message); }
    rc = ll_get_data(step, LL_StepWallClockLimitHard, &WCLimit); /* get hard WC limit */
    if (rc != 0) {sprintf(Message, "LL_StepWallClockLimitHard failed"); report(rc, Message); }
    if (state == STATE_RUNNING) { /* job running... */
        LocalTime = localtime(&endtime); /* convert dispatch time to time structure */
    } else { /* job not running... */
        LocalTime = localtime(&SubmitTime); /* convert submit time to time structure */
    }
}
strftime(TextTime,smax, "%m/%d/%y-%H:%M:%S",LocalTime); /* format
time for output */
    sprintf(TempBuff, " %s",TextTime);
    strcat(OutBuff,TextTime); /* copy into output buffer */

    endtime += WCLimit; /* add WC time to figure out completion time */
    if (state == STATE_RUNNING) { /* if job running... */
        LocalTime = localtime(&endtime); /* Convert and format END time */
        strftime(TextTime,smax, "%m/%d/%y-%H:%M:%S",LocalTime);
        sprintf(TempBuff, " %s",TextTime);
    } else { /* job not running... */
        strftime(TextTime,smax, "%H:%M:%S",LocalTime); /**/
        sprintf(TempBuff, "%d",WCLimit/60); /* WC time in minutes */
    } /* Request the objects from the Negotiator daemon */
    rc = ll_get_data(step,LL_StepMachineCount, &numnodes);
    if (rc != 0) {sprintf(Message, "LL_StepMachineCount failed");
        report(rc, Message); }

    /* Request the objects from the Negotiator daemon */
    rc = ll_get_data(step,LL_StepGetFirstNode, &node);
    if (rc != 0) {sprintf(Message, "LL_StepGetFirstNode failed");
        report(rc, Message); }
    rc = ll_get_data(node,LL_NodeMinInstances, &minnodes); /* get
        minnode setting..*/
    if (rc != 0) {sprintf(Message, "LL_NodeMinInstances failed");
        report(rc, Message); }
    rc = ll_get_data(node,LL_NodeInitiatorCount, &numinit); /* get
        initiator count..*/
    if (rc != 0) {sprintf(Message, "LL_NodeInitiatorCount failed");
        report(rc, Message); }
    numtasks = minnodes * numinit;
    /**
        printf("minnodes=%d, initcount=%d\n",minnodes, numinit);
    **/
    sprintf(TempBuff, " %d",numtasks);
    strcat(OutBuff,TempBuff); /* copy into output buffer */

    if(ll_get_data(job,LL_JobCredential, &cred)){ /* go get user name
        and print it */
        printf("Couldn't get credential object.\n");
    } else {
        if(ll_get_data(cred,LL_CredentialUserName, &name)==0) {
Appendix A. Tools

```c
sprintf(TempBuff," %s",name);  
strcat(OutBuff,TempBuff); /* copy into output buffer */  
free(name);
} else {  
    sprintf(Message, "LL_CredentialUserName failed - can't get  
user name");  
    report(rc, Message);
}

rc = ll_get_data(step,LL_StepJobClass, &JobClass); /* get job class */  
if (rc != 0) {sprintf(Message, "LL_StepJobClass failed");  
report(rc, Message);  
sprintf(TempBuff," %s", JobClass); /* Get Job Class */  
strcat(OutBuff,TempBuff); /* copy into output buffer */  
printit(state,OutPrint,OutBuff);

free(JobClass);

/*** The code below prints out a line of requested resources (if it exists) ***/

rc = ll_get_data(step,LL_StepComment, &Comment); /* get comment -  
resources for job */  
if (rc != 0) {sprintf(Message, "LL_StepComment failed"); report(rc,  
Message);  
if (*Comment != NULL) { /* if there's something in the comment... */  
    sprintf(OutBuff,"Resources: %s",Comment); /* jam into buffer */  
Comment=NULL; /* reset the output buffer */  
    printit(state,OutPrint,OutBuff); /* go print the output buffer */  
}

free(Comment);

/*** This next section prints the node list for running jobs ***/

    if (state == STATE_RUNNING && (OutPrint==RUNNING || OutPrint==ALL)  
&& Verbose==TRUE) {  
        /* print the node information */  
        sprintf(OutBuff,"Nodes: "); /* now, we'll print the node list */  
        for (j=0 ; j<numnodes ; ++j) { /* for the list of nodes */  
        if (j==0) { /* if we're reading the first record... */  
            rc = ll_get_data(step,LL_StepGetFirstMachine, &machine); /*  
get first machine record */  
        if (rc != 0) {sprintf(Message, "LL_StepGetFirstMachine  
failed"); report(rc, Message);  
            rc = ll_get_data(machine,LL_MachineName, &name); /* get the  
first name */
```
if (rc != 0) {sprintf(Message, "LL_MachineName failed");
report(rc, Message);}
else { /* we're reading records 2 thru N...*/
rc = ll_get_data(step, LL_StepGetNextMachine, &machine); /*
get next machine info */
if (rc != 0) {sprintf(Message, "LL_StepGetNextMachine
failed"); report(rc, Message);
rc = ll_get_data(machine, LL_MachineName, &name);
if (rc != 0) {sprintf(Message, "LL_MachineName failed
while reading machine list"); report(rc, Message);}
}
for (i=0 ; i<B_SIZE ; ++i) *(LocalMach+i)=0; /* buff clear */
TempPtr = strchr(name, HEXDOT); /* hack off the extended
machine ID */
strncpy(LocalMach, name, TempPtr-name);
sprintf(TempBuff, "%s\n", LocalMach); /* print it */
strcat(OutBuff, TempBuff);
free(name);
printf("%s\n", OutBuff);
}
rc=ll_get_data(job, LL_JobGetNextStep, &step);
if (rc != 0) {sprintf(Message, "LL_JobGetNextStep failed");
report(rc, Message);}
} /* End of processing for one job */

job = ll_next_obj(queryObject); /* get the record for the next job */
} /* End of loop on all the job steps */
} /* end of loop on all the jobs */

/* free objects obtained from Negotiator */
rc = ll_free_objs(queryObject); /**/
if (rc != 0) {sprintf(Message, "LL_freeobjs failed"); report(rc, Message);}
/* free query element */
rc = ll_deallocate(queryObject); /**/
if (rc != 0) {sprintf(Message, "LL_deallocate failed"); report(rc,
Message);}
exit(0);
} /* end main */
**ll_get_nodes.c**

Example A-2 is the source code for ll_get_nodes tool.

*Example: A-2  Source code for ll_get_nodes tool

/*****************************/
Code to test data query api

2/07/00 - This version mimics api_get_node / api_get_queue operation

#include "/usr/lpp/LoadL/full/include/llapi.h"/* Get lib calls, data types and
structures */
#include <time.h>
#include <string.h>

#define MATCH 0
#define B_SIZE 1000
#define TRUE 1
#define FALSE 0

int report(int rc, char *Message) /* subroutine to exit on error */
printf("%s. rc = %d\n",Message,rc);
exit (rc);
} /* end of report subroutine */

int main(int argc, char *argv[])
{
/* Define local variables */

LL_element *queryObject=NULL, *machine=NULL;
int i,j,rc,num,err;
int MachineMemory, MachineMaxTasks, RunningJobs;
char *MachineName, *OneFeature, *State, *OneStep;
char **MachineFList, **StepList;
char *OutBuff, *TempBuff;
char *Message;

TempBuff = (char *)malloc( B_SIZE);
OutBuff = (char *)malloc(5 * B_SIZE);
Message = (char *)malloc(B_SIZE);

/* Initialize the query for machines */
queryObject = ll_query(MACHINES);
/* I want to query all machines */
rc = ll_set_request(queryObject, QUERY_ALL, NULL, NULL);
if (rc != 0) { printf(Message, "ll_set_request failed"); report(rc, Message); } 

/* Request the objects from the Negotiator daemon */
machine = ll_get_objs(queryObject, LL_CM, NULL, &num, &err);

/* Did we get a list of objects ? */
if (machine == NULL) {
  printf(" err = \d\n", err);
  /* no machines to report, don't print anything...*/
} else {
  /* Loop through the list and process */
  while (machine) {
    rc = ll_get_data(machine, LL_MachineName, &MachineName); /* get the machine name*/
    if (rc != 0) (printf(Message, "LL_MachineName failed"); report(rc, Message));

    sprintf(OutBuff, "%s:\", MachineName);
    free(MachineName);

    rc = ll_get_data(machine, LL_MachineRealMemory, &MachineMemory); /* get the machine memory*/
    if (rc != 0) (printf(Message, "LL_MachineRealMemory failed"); report(rc, Message));
    sprintf(TempBuff, "%d;", MachineMemory); /* save in output buffer */
    strcat(OutBuff, TempBuff); /* copy into output buffer */

    rc = ll_get_data(machine, LL_MachineFeatureList, &MachineFList); /* get the machine features*/
    if (rc != 0) {printf(Message, "LL_MachineFeatureList failed"); report(rc, Message);}
    OneFeature = *MachineFList; /* get first feature */
    i = 0;
    while (OneFeature != NULL) { /* while more features...*/
      sprintf(TempBuff, "%s;", OneFeature); /* save in output buffer */
      strcat(OutBuff, TempBuff); /* copy into output buffer */
      ++i;
      OneFeature = *(MachineFList + i); /* get one feature.. */
    }
    /* add a terminator once last feature written */
    sprintf(TempBuff, ":");
    strcat(OutBuff, TempBuff); /* copy into output buffer */
    free(OneFeature);
    free(MachineFList);
rc = ll_get_data(machine, LL_MachineMaxTasks, &MachineMaxTasks); /* get the machine Max Tasks*/
    if (rc != 0) {sprintf(Message, "LL_MachineMaxTasks failed");
        report(rc, Message); }
    sprintf(TempBuff, "%d:", MachineMaxTasks); /* save in output buffer */
        strcat(OutBuff, TempBuff); /* copy into output buffer */

rc = ll_get_data(machine, LL_MachineStartdRunningJobs, &RunningJobs); /* get the machine Running jobs*/
    if (rc != 0) {sprintf(Message, "LL_MachineStartdRunningJobs failed");
        report(rc, Message); }
    sprintf(TempBuff, "%d:", RunningJobs); /* save in output buffer */
        strcat(OutBuff, TempBuff); /* copy into output buffer */

rc = ll_get_data(machine, LL_MachineStartdState, &State); /* get the StartD state*/
    if (rc != 0) {sprintf(Message, "LL_MachineStartdState failed");
        report(rc, Message); }
    sprintf(TempBuff, "%s:", State); /* save in output buffer */
        strcat(OutBuff, TempBuff); /* copy into output buffer */
    free(State);

rc = ll_get_data(machine, LL_MachineStepList, &StepList); /* get the machine Step List*/
    if (rc != 0) {sprintf(Message, "LL_MachineStepList failed");
        report(rc, Message); }
    OneStep = *StepList; /* get first step */
        i = 0;
    while (OneStep != NULL) { /* while more steps...*/
        sprintf(TempBuff, "%s," , OneStep); /* save in output buffer */
            strcat(OutBuff, TempBuff); /* copy into output buffer */
                ++i;
        OneStep = *(StepList + i); /* get one feature.. */
        }
    free(OneStep);
    free(StepList);

printf("\n", OutBuff);
machine = ll_next_obj(queryObject); /* get the record for the next machine */
}
/* free objects obtained from Negotiator */
rc = ll_free_objs(queryObject); /***/
    if (rc != 0) {sprintf(Message, "LL_freeobjs failed"); report(rc, Message); } /* free query element */
rc = ll_deallocate(queryObject); /***/
    if (rc != 0) {sprintf(Message, "LL_deallocate failed"); report(rc, Message); }
Example program hist.c

Example A-3 is an example of the hist.c program.

Example: A-3  llhist.c
/* Desc: Converts LoadL binary history file into ASCII */
#include "llhist.h"

int main(int argc, char **argv) {
    char *filename = NULL;
    struct stat st;

    /* Check arguments */
    if (argc != 2) {
        fprintf(stderr, "Usage: ll_hist <global history file>

        exit (1);
    }

    /* Get name of history file */
    filename = *++argv;

    /* Double check! */
    if (filename == NULL) {
        fprintf(stderr, "Usage: ll_hist <global history file>

        exit (1);
    }

    /* Check file exists */
    if (lstat(filename, &st) < 0) {
        perror(filename);
        exit(1);
    }

    /* Print header */
    printf("#Job Name:User:Group:Uid:Gid:Submit
/ * Process history records */
if (GetHistory(filename, process_record, LL_JOB_VERSION) < 0) {
    fprintf(stderr, "An error has occurred opening the history file %s\n",
            filename);
    exit(1);
}
return (0);

Example program llacct.c

Example A-4 is an example of the llacct.c program.

Example: A-4 llacct.c
/* Desc: Util code used by llhist.c */
#include "llhist.h"
/* Invoked by GetHistory() to process LoadL history record */
int process_record(LL_job *job_ptr) {
    int i, x, procnum, usr_secs, sys_secs;
    LL_job_step *step;
    printf("%s:%s:%s:%d:%d:%s:%d:", job_ptr->job_name,
           job_ptr->owner,
           job_ptr->groupname,
           job_ptr->uid,
           job_ptr->gid,
           job_ptr->submit_host,
           job_ptr->steps);

    step = job_ptr->step_list[0];
    for (i=0; i<job_ptr->steps; i++) {
        procnum = step->num_processors;
        printf("%s:%s:%s:%s:%s:%s:", step->requirements,
               (step->flags & LL_CHECKPOINT)?("yes"):("no"),
               (step->flags & LL_RESTART)?("yes"):("no"),
               (step->flags & LL_STEP_PARALLEL)?("parallel"):("serial"),
               step->stepclass);

Appendix A. Tools 183
/* WallClock Limit , No. of Procs, LoadL Job ID, Image Size */
printf("%d:%d:%s.%d.%d:%d:",
step->limits.hard_wall_clock_limit,
procnum,
strtok(step->id.from_host,"."),
step->id.cluster,
step->id.proc,
step->image_size);

/* Queue Date, Start Date, Completion Date */
printf("%d:%d:%d:", step->q_date, step->dispatch_time,
step->completion_date);

/* Acquired Nodes */
for (x=0; x<procnum; x++) {
    printf("%s%s", step->processor_list[x], (x < (procnum-1))?
(","):(""));
}

/* Job User Time (s), Job System Time (s), Total Job Time (s) */
usr_secs = step->usage_info.step_rusage.ru_utime.tv_sec;
sys_secs = step->usage_info.step_rusage.ru_stime.tv_sec;
printf("%d:%d:%d:", usr_secs, sys_secs, (usr_secs + sys_secs));

/* Job Completion Status */
printf("%s\n", getStatus(step->status));
step++;

return(0);
}

char *getStatus(int val) {
    char *status;
    switch(val) {
    case 0: status="idle"; break;
    case 1: status="starting"; break;
    case 2: status="running"; break;
    case 3: status="removed"; break;
    case 4: status="completed"; break;
    case 5: status="hold"; break;
    case 6: status="deferred"; break;
    case 7: status="submission error"; break;
    case 8: status="vacated"; break;
    }
case 9: status="not running"; break;
case 10: status="not queued";
}
return status;
}

Header and Makefiles for getting history data

Example A-5 is the header file for llhist.h, while Example A-6 is the header file for
the history data makefile.

Example: A-5  llhist.h header
/* Desc: Header file for llacct */

#ifndef __LLHIST__
define __LLHIST__

#include <sys/resource.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <stdio.h>
#include <string.h>
#include "llapi.h"

extern int GetHistory(char *filename, int process_record(LL_job *job_ptr), int
version);
int process_record(LL_job *job_step);
char *getStatus(int val);

#endif /*__LLHIST__*/

Example: A-6  Makefile
CC=xlc
LOADL_DIR=/usr/lpp/LoadL/full
LIB_DIR=$(LOADL_DIR)/lib
INC_DIR=$(LOADL_DIR)/include
LIBS=llapi
OBJS=llhist.o llhist_utils.o
CFLAGS=-O2
LDFLAGS=-L$(LIB_DIR) -l$(LIBS)

all: llhist

llhist: $(OBJJS)
    $(CC) -o $@ $(OBJJS) $(LDFLAGS)
llhist.o: llhist.c
   $(CC) $(CFLAGS) -c $? -I$(INC_DIR)

llhist Utils.o: llhist Utils.c
   $(CC) $(CFLAGS) -c $? -I$(INC_DIR)

clean:
   /usr/bin/rm -f llhist *.o
Concise reference

This appendix can be used as a concise reference for LoadLeveler environment and provides the following:

- Environment variables
- Job command variables
- Job command file keywords
- Jobs status variables
- Daemon states and their meanings
- Debug flags
- Command reference
LoadLeveler environment variables

Example settings of environment variables set for a job:

LOADL_STEP_COMMAND=test3.cmd
LOADL_STEP_ACCT=a00001
LOADL_STEP_GROUP=group1
LOADL_PID=17574
LOADL_JOB_NAME=test2
LOADL_STEP_OWNER=peterm
LOADL_STEP_NAME=one
LOADLBATCH=yes
LOADL_STEP_NICE=0
LOADL_STEP_TYPE=SERIAL
LOADL_STEP_INITDIR=/u/peterm/LL_RESIDENCY/TESTS/TEST6
LOADL_STARTD_PORT=9611
LOADL_STEP_ERR=test2.233.0.err
LOADL_PROCESSOR_LIST=node15.stc.uk.ibm.com
LOADL_STEP_IN=/dev/null
LOADL_STEP_OUT=test2.233.0.out
LOADL_ACTIVE=2.2.0.8
LOADL_STEP_ID=node09.stc.uk.ibm.com.233.0
LOADL_STEP_ARGS=
LOADL_STEP_CLASS=express

LoadLeveler variables

Arch
ConsumableCpus
ConsumableMemory
ConsumableVirtualMemory
Cpus
CurrentTime
CustomMetric
Disk
domain/domainname
EnteredCurrentState
host/hostname
KeyboardIdle
LoadAvg
Machine
Memory
MasterMachPriority
OpSys
QDate
Speed
State
tilde
UserPrio
VirtualMemory

tm_hour
tm_min
tm_sec
tm_isdst
tm_mday
tm_wday
tm_yday
tm_mon
tm_year
tm4_year

Job command file variables

jobid = cluster
stepid = process
base_executable
host = hostname
domain
executable
class
comment
job_name
step_name
schedd_host
schedd_hostname

Job command file keywords

account_no
arguments
blocking
checkpoint
class
comment
core_limit
cpu_limit
data_limit
dependency
environment
error
executable
file_limit
group
hold
image_size
initialdir
input
job_cpu_limit
job_name
job_type
max_processors
min_processors
network
node_usage
notification
notify_user
output
parallel_path
preferences
requirements
resources
restart
rss_limit
shell
stack_limit
start_date
step_name
step_name
task_geometry
tasks_per_node
total_tasks
user_priority
wall_clock_limit

Job states

   Canceled             CA
   Completed             C
   Complete Pending      CP
   Deferred              D
   Idle                  I
   NotQueued             NQ
   Not Run               NR
   Pending               P
   Rejected              X
   Reject Pending        XP
   Removed               RM
Remove Pending        RP
Running               R
Starting              ST
Submission Error      SX
System Hold           S
System User Hold      HS
Terminated            TX
User Hold             H
Vacated               V
Vacate Pending        VP
Checkpointing         CK
Preempted             E
Preempt Pending       EP
Resume Pending        MP

Daemon States

Here is the list of daemon states for the Schedd and Start daemons.

Schedd

Available      Avail
Draining       Drning
Drained        Drned
Down

Startd

Busy
Down
Drained        Drned
Draining       Drning
Flush
Idle
None
Running       Run
Suspend       Suspnd

Admin file

label: type = machine
  adapter_stanzas
  alias
  central_manager
cpu_speed_scale
dce_host_name
machine_mode
master_node_exclusive
max_adapter_windows
max_jobs_scheduled
name_server
pvm_root
pool_list
resources
schedd_fenced
schedd_host
spacct_excluse_enabled
speed
submit_only

label: type = user
account
default_class
default_group
maxidle
maxjobs
maxqueued
max_node
max_processors
max_total_tasks
priority
total_tasks

label: type = class
admin
class_comment
default_resources
exclude_groups
exclude_users
include_groups
include_users
master_node_requirement
maxjobs
max_node
max_processors
nice
NQS_class
NQS_submit
NQS_query
priority
total_tasks
core_limit
cpu_limit
cpu_limit
data_limit
file_limit
job_cpu_limit
rss_limit
stack_limit
wall_clock_limit
chkp_dir
ckpt_time_limit
execution_factor
max_total_tasks

class: type = group
    admin
    exclude_users
    include_users
    maxidle
    maxjobs
    maxqueued
    max_node
    max_processors
    priority
    total_tasks
    max_total_tasks

class: type = adapter
    adapter_name
    css_type
    interface_address
    interface_name
    network_type
    switch_node_number
    multilink_address
    multilink_list

Debug flags

D_ACCOUNT
D_AFS
D_DAEMON
D_DBX
D_DCE
D_EXPR
D_FULDEBUG
D_JOB
D_KERNEL
D_LOAD
D_LOCKING
D_MACHINE
llctl subcommands

- purge
- capture
- drain
- flush
- purgeschedd
- reconfig
- recycle
- resume
- start
- stop
- suspend
- version

llstatus -f /-r field selectors

- %a  Hardware architecture
- %act  Number of jobs dispatched by the schedd on this machine
- %cm  Custom Metric Value
- %cpu  Number of CPUs on this machine
- %d  Available disk space in the LoadLeveler execute directory
- %i  Number of seconds since last keyboard or mouse activity
- %inq  Number of jobs in queue that were scheduled from this machine
- %l  Berkeley one-minute load average
- %m  Physical memory on this machine
- %mt  Maximum number of tasks that can run simultaneously on this machine
- %n  Machine name
- %o  Operating system on this machine
- %r  Number of jobs running on this machine
- %sca  Availability of the schedd daemon
- %scs  State of the schedd daemon
- %sta  Availability of the startd daemon
- %sts  State of the startd daemon
- %v  Available swap space of this machine
Ilq -f/-r field selectors

%a Account number
%c Class
%cc Completion code
%dc Completion date
%dd Dispatch date
%dh Hold date
%dq Queue date
%gl LoadLeveler group
%gu UNIX group
%h Host
%id Step ID
%is Virtual image size
%jn Job name
%jt Job type
%mh Number of hosts allocated to the job
%o Job owner
%p User priority
%sn Step name
%st Status
Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Publications

For information on ordering these publications, see “How to get IBM Redbooks” on page 197.

- Loadleveler V2R2 for AIX: Using and Administering, SA22-7311
- Parallel Environment for AIX: Operations and Use, Vol 1, SA22-7425
- Parallel Environment for AIX: Operations and Use, Vol 2, SA22-7426

Referenced Web sites

These Web sites are also relevant as further information sources:

- http://supercluster.org
- http://supercluster.org/maui

How to get IBM Redbooks

Search for additional Redbooks or redpieces, view, download, or order hardcopy from the Redbooks Web site:

ibm.com/redbooks

Also download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

Redpieces are Redbooks in progress; not all Redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.
IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.
Special notices

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

This document has not been subjected to any formal review and has not been checked for technical accuracy. Results may be individually evaluated for applicability to a particular installation. You may discuss pertinent information from this document with a customer, and you may abstract pertinent information for presentation to your customers. However, any code included is for internal information purposes only and may not be given to customers. If included code is identified as incidental programming, its use must conform to the guidelines in the relevant section of the sales manual.

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere., The Power To Manage., Anything. Anywhere., TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Kjøbenhavns Sommer - Tivoli A/S.

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.
UNIX is a registered trademark in the United States and other countries licensed exclusively through The Open Group.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.
Index

Symbols
/etc/environment 53
/etc/LoadL.cfg 3
/etc/WLM/current 73
/etc/WLM/LoadL 73
/u/loadl 2
/usr/bin/poe 8
/usr/lib/libpthreads.a 95
/usr/lpp/LoadL/full/bin 5
/usr/lpp/LoadL/full/include 6
/usr/lpp/LoadL/full/include/llapi.h 116
/usr/lpp/LoadL/full/lib 6
/usr/lpp/LoadL/full/samples/llphist 144
/usr/lpp/ssp/css/samples/lapi 72
/var/loadl 11

A
A DETAIL 140
A OFF 140
A ON 140
A VALIDATE 140
Accounting API 116
adapter stanza 24
Adapter striping 17
adapters 3
AIX WLM 57
AIX Workload Manager 73
Allocate 2
allocation 39
API 6, 115
   Accounting 6
   Checkpointing 6
   Data Access 6
   Parallel Job 6
   Query 6
   Submit 6
   Workload Management 6
Application Programming Interfaces 6
Availability 2

B
BACKFILL 45

Batch 2, 24
Batch parallel jobs 57
Blocking 18, 60

C
C 6
C++ 6
Central Manager 4, 73
Checkpoint
   ckpt() 24
   Fileset
      bos.cpr 24
   restart 24
   System initiated 81
   User initiated 81
   User-initiated 24
   checkpoint 24
   file 24
   Checkpoint / Restart 17
   Checkpointing 81
cckpt_cleanup_interval 85
cckpt_cleanup_program 85
Class 3, 39
class stanza 40
classSysprio 98
Cluster 2
Command
   installlp 10
   llacctmrg 5, 142
   lcancel 5
   llckpt 83
   llclass 5
   llCtl 5, 15
   llextSDR 5
   llimit 5, 12
   llmatrix 5, 109
   llpreempt 23
   llq 4
   llstatus 4, 5
   llsubmit 4, 5, 35
   llspp 10
   supper 11
   wlmstat 75
xloadl_so 6
xoadl 5
commands
dsh 11
Configuration
global file 3
master file 3
configuration files 2
Consumable 73
Consumable Resources 17, 18, 19, 57
ConsumableCpus 19
ConsumableMemory 19, 75, 77
ConsumableVirtualMemory 19
context switch 23, 108
Context Switching 112
CONTINUE 51
cpu_limit 46
CSS 22
Customizing ltq 151

D
daemon 21
Daemons 2, 4
data access 117
Data access API 116
Daytime 52
DCE 21
DCE support 17
Debug 63
Directives 7
Directory
execute 11
home directory 12
local 11
log 11
release 13
spool 11
Distributed 2
drain 43

e
ENFORCE_RESOURCE_SUBMISSION 19
ENFORCE_RESOURCE_USAGE 19
enforcement 19
Epilog 3, 36, 53
Evening 52
EveningClass 52
execution_factor 108, 111
external scheduler 6

F
File Collection 11
filessets 2
filesystem 12
floating resources 19

G
Gang central manager 108
Gang matrix 5, 22
GANG scheduler 22
Gang-scheduling 17
Global history 140
GPFS 24, 54, 94
group 3
Group Services 4, 20
GroupRunningJobs 49
GroupSysPrio 98
GroupTotalJobs 49
gsmonitor 4, 17, 20
GSMONITOR_RUNS_HERE 20
GUI 5

H
history 3
History files 140
home directory 2

I
Install 9
/var/load 11
Control workstation 10
create group 11
create user 11
dsh 11
filessets 10
installp command 10
llimit 12
lslpp 10
product 10
interactive 24
interactive parallel job 57, 61
IP 65
IP striping 24
J

JFS 11
Job 2, 4
  batch 2
equivalent 7
job command file 2
parallel 2
serial 2
Job command file 2, 38
$(cluster) 38
$(domain) 38
$(host) 38
$(hostname) 38
$(jobid) 38
$(process) 38
$(stepid) 38
job command file 7
Job filter 35
Job Filters 27
job restart 82
job step 28
Job steps 27
Job submission filter 35
JOB_ACCT_Q_POLICY 140
JOB_LIMIT_POLICY 141
Jobs
  dependencies 31
drain 43
filter 35
independent 31
submit 35

K

Kerberos 15
kernel extension 20
Keyword
  ACCT 140
  ACCT_VALIDATION 143
  CC_NOTRUN 30
  CC_REMOVED 30
  CHECKPOINT 82
  ckpt_dir 84
  ckpt_time_limit 84
cpu_speed_scale 145
  CUSTOM_METRIC 102
  CUSTOM_METRIC_COMMAND 102
default_interactive_class 61
  FLOATING_RESOURCES 74
  GANG_MATRIX_BROADCAST_CYCLE 111
  GANG_MATRIX_REORG_CYCLE 110
  GANG_MATRIX_TIME_SLICE 23, 110
  GLOBAL_HISTORY 140
  JOB_EPILOG 53
  JOB_PROLOG 53
  JOB_USER_EPILOG 53
  JOB_USER_PROLOG 53
max_jobs_scheduled 48
MAX_STARTD_LOG 161
MAX_STARTERS 41
NEGOTIATOR_PARALLEL_DEFER 105
NEGOTIATOR_PARALLEL_HOLD 105
NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL 50
node 58
node_usage 79
NotQueued 31
POLLING_FREQUENCY 52
PREEMPT_CLASS 23, 109
PROCESS_TRACKING 20, 106
PROCESS_TRACKING_EXTENSION 20
restart_from_ckpt 87
SAVELOGS 162
SCHEDULE_BY_RESOURCES 73
SCHEDULER 104, 105, 106
SCHEDULER_API 104
spacct_excluse_enable 147
speed 145
START_CLASS 40, 110
STARTD_LOG 161
SUBMIT_FILTER 35
task_geometry 59
tasks_per_node 58
total_tasks 58
wall_clock_limit 105
KILL 51

L

LAPI 72
limits 45
llacctmrg 5
llapi.h 6
llcancel 5
llckpt 83
llclass 5
llct 5
llextSDR 5, 14
Ifavorjob 5
Ilhold 5
Ilinit 5
Ilmatrix 5, 109
Ilmodify 5
Ilpreempt 5, 23
Ilprio 5
Ilq 4, 5, 21
Ilstatus 4, 5, 21
Ilsubmit 4, 5
Ilsummary 5
LOAD_STEP_ID 36
LoadAvg 102
LOADL_ACTIVE 36
LoadL_admin 3, 14
LOADL_CONFIG 3
LoadL_config 3, 15
LoadL_config.local 3
LoadL_GSmonitor 4, 20
LOADL_JOB_STEP_EXIT_CODE 54
LoadLkbd 4
LoadL_master 4
LoadL_negotiator 4
LOADL_PROCESSOR_LIST 54
LOADL_PROLOG_EXIT_CODE 54
LoadL_schedd 4
LoadL_startd 4
LoadL_starter 4
LOADL_STEP_COMMAND 36
LOADL_STEP_NAME 37
LOADL_STEP_OWNER 36
LOADL_USER_PROLOG_EXIT_CODE 54
LoadLeveler 2
Local history 140
LOCAL_CONFIG 13
log 3
Log files 161

M
Machine 2
allocate 2
pool 2
machine stanza 14
MACHPRIO 58
Mail Notification 155
manage 2
Management 4
Managing classes 27
Managing Job execution 27
Managing Job queue 27
Managing limits 27
Maui 115
MAX_CKPT_INTERVAL 83
max_node 48
max_processors 48
MAX_STARTERS 110
maxidle 48
maxjobs 48
maxqueued 48
Memory
real 18
virtual 18
Message passing library 8
MIN_CKPT_INTERVAL 83
MINUTE 50
MP_CKPTDIR 87
MP_CKPTFILE 87
MP_EUIDEVICE 61
MP_EUILIB 61
MP_HOSTFILE 18
MP_INFOLEVEL 63
MP_LLFILE 66, 67
MP_SHARED_MEMORY 8
MP_WAIT_MODE 80
MPI 8, 19
multiprogramming 23
multi-threaded 18
multi-threads 73

N
Negotiator 50
negotiator 4
network 3
network_type 68
nice value 40
not_shared 79

O
OpenMP 19, 73, 95

P
Parallel 2, 8
parallel task 18
parse 35
Pathname 3
Perl script 35
POE 61, 88
Polling interval 20
Pool 2
Preemption 17, 23, 113
PrimeClass 52
priority 40
process 18, 20
process limits 45
Process tracking 17, 20
Product 10
Prolog 3, 36, 53
pulldown menus 148

Q
QDate 98
query 6
Query API 116
queue 3, 47
queuing priorities 98

R
Redbooks Web site 197
Contact us xvi
Resource 2
Resources 4, 39
rlim_infinity 46
Run-time environment variables 27
Runtime variables 36

S
Schedule 2
Scheduler 73
BACKFILL 45, 97
default 104
external 6
GANG 22, 97, 106
execution 23
gang 22
internal 97
LL_DEFAULT 97
SDR 5
Serial 2
Serial checkpoint API 116
Server 2
share 76
shared 79
slice_not_shared 79
SMP 18
SMP nodes 57
snapshot 82
SP switch 72
SP_Switch 22
SP_Switch2 22
Special Resource Scheduler 132
spool 3
SRS 115
START 51
start 15
startd 4
starter 4
Startup 4
Status
  checkpointing 42
draining 43
Running 42
Starting 42
Submit API 116
Submit-only 2, 4
supper 11
SUSPEND 51
suspend 52
switch adapters 24
symbolic links 13
SYSPRIO 49, 102

T
Task Assignment 17, 57
Task geometry 59, 87
tasks_per_node 18
threaded MPI 95
TMPDIR 54
total_tasks 18, 49

U
user 3
User space 22, 69
UserPrio 98
UserQueuedJobs 49
UserRunningJobs 49
UserSysPrio 98
UserTotalJobs 49
utilization 2
V
VACATE 51
verification 35

W
wall_clock_limit 46
WeekDay 52
WeekendClass 52
WLM 19
WLM class 73
WLM configuration 19
WLM integration 17, 18
WLM resource 75
WORKDIR 54
Workload Management 19
Workload management API 116
Workstation 2

X
xloadl 5, 148
IBM LoadLeveler is a very effective tool for managing serial and parallel jobs over a cluster of servers. This redbook describes the new tools and functions supported by LoadLeveler Version 3.1, such as new schedulers, checkpointing of jobs, AIX Workload Manager integration, and so on, and includes example scenarios. Performance tuning of large clusters, and the use of external schedulers, are also featured in this book.

This redbook will help LoadLeveler administrators by providing a quick guide to installation procedures; we also discuss how to exploit various LoadLeveler features with sample scenarios.