Univa, An Altair Company

Grid Engine Documentation

Grid Engine Users's Guide

Author:
Altair Engineering

Version:
8.7.0

September 6, 2021

© 2021 ALTAIR ENGINEERING, INC. ALL RIGHTS RESERVED.

WE ARE CURRENTLY LISTED ON NASDAQ AS ALTR. UNIVA IS AN ALTAIR COMPANY

Contents

1 Overview of Basic User Tasks 1

2 A Simple Workflow Example 1

3 Displaying Altair Grid Engine Status Information 5
3.1 Cluster Overview . . . 5
3.2 Hosts and Queues . . . 5
3.3 Requestable Resources . . . 8
3.4 User Access Permissions and Affiliations . . . 10

4 Submitting Batch Jobs 13
4.1 What is a Batch Job? . . . 13
4.2 How to Submit a Batch Job . . . 13
4.2.1 Example 1: A Simple Batch Job . . . 13
4.2.2 Example 2: An Advanced Batch Job . . . 14
4.2.3 Example 3: Another Advanced Batch Job . . . 14
4.2.4 Example 4: A Simple Binary Job . . . 15
4.3 Specifying Requirements . . . 15
4.3.1 Request Files . . . 16
4.3.2 Requests in the Job Script . . . 17

5 Using Job Classes to Prepare Templates for Jobs 17
5.1 Examples Motivating the Use of Job Classes . . . 18
5.2 Defining Job Classes . . . 19
5.2.1 Attributes describing a Job Class . . . 20
5.2.2 Example 1: Job Classes - Identity, Ownership, Access . . . 22
5.2.3 Attributes to Form a Job Template . . . 22
5.2.4 Example 2: Job Classes - Job Template . . . 25
5.2.5 Access Specifiers to Allow Deviation . . . 26
5.2.6 Example 3: Job Classes - Access Specifiers . . . 28
5.2.7 Different Variants of the same Job Class . . . 29
5.2.8 Example 4: Job Classes - Multiple Variants . . . 30
5.2.9 Enforcing Cluster Wide Requests with the Template Job Class . . . 31

5.3 Relationship Between Job Classes and Other Objects 33
5.3.1 Resources Available for Job Classes 33
5.3.2 Defining Job Class Limits 34
5.3.3 JSV and Job Class Interaction 34
5.4 Commands to Adjust Job Classes 35
5.4.1 Creating, Modifying and Deleting Job Classes 35
5.4.2 States of Job Classes 36
5.5 Using Job Classes to Submit New Jobs 37
5.6 Example: Submit a Job Class Job and Adjust Some Parameters 38
5.7 Status of Job Classes and Corresponding Jobs 39

6 Monitoring and Controlling Jobs **40**
6.1 Getting Status Information on Jobs 40
6.2 Deleting a Job 42
6.3 Re-queuing a Job 43
6.4 Modifying a Waiting Job 43
6.4.1 Altering Job Requirements 44
6.5 Changing Job Priority 44
6.6 Obtaining the Job History 45

7 Other Job Types **46**
7.1 Array Jobs 46
7.2 Interactive Jobs 48
7.2.1 qrsh and qlogin 49
7.2.2 qmake 49
7.2.3 qsh 50
7.3 Parallel Jobs 50
7.3.1 Parallel Environments 51
7.3.2 Submitting Parallel Jobs 53
7.4 m_numa_nodes Amount of NUMA nodes on the execution host. 60
7.4.1 Memory Allocation Strategy round_robin 61
7.4.2 Memory Allocation Strategy cores and cores:strict 62
7.4.3 Memory Allocation Strategy nlocal 64
7.5 Checkpointing Jobs 65

7.5.1 User-Level Checkpointing 66
7.5.2 Kernel-Level Checkpointing 66
7.5.3 Checkpointing Environments 66
7.5.4 Submitting a Checkpointing Job 67
7.6 Immediate Jobs 68
7.7 Reservations 68
7.7.1 Advance Reservations 69
7.7.2 Standing Reservations 72
7.8 Jobs using Docker Containers 79
7.8.1 Running a sequential job in a Docker container 80
7.8.2 Running a parallel Job in Docker containers 83
7.8.3 Running MPI jobs in Docker containers 84
7.8.4 Running an array Job in Docker containers 84
7.8.5 Running a Job in a Docker image that is not available locally 85
7.8.6 Using placeholders to dynamically define Docker options 85
7.8.7 Support for nvidia-docker 2.0 86

8 Getting a Consistent View onto the System by Using Sessions 86
8.1 Communication with Altair Grid Engine without using Sessions 87
8.2 Using sessions to communicate with the system 87

9 Submission, Monitoring and Control via an API 89
9.1 The Distributed Resource Management Application API (DRMAA) 89
9.2 Basic DRMAA Concepts 89
9.3 Supported DRMAA Versions and Language Bindings 90
9.4 When to Use DRMAA 90
9.5 Environment Variable Influences 90
9.6 Examples 90
9.6.1 Building a DRMAA Application with C** 90
9.6.2 Building a DRMAA Application with Java 93
9.7 Further Information 95

10 Advanced Concepts **95**
10.1 Job Dependencies . . . 96
10.1.1 Examples . . . 96
10.2 Using Environment Variables . . . 98
10.3 Using the Job Context . . . 100
10.4 Transferring Data . . . 101
10.4.1 Transferring Data within the Job Script . . . 101
10.4.2 Using Delegated File Staging in DRMAA Applications . . . 102
10.5 Manual, Semi-Automatic and Automatic Preemption . . . 103
10.5.1 Preemption Terms . . . 104
10.5.2 Preemption Trigger and Actions . . . 104
10.5.3 Manual Preemption . . . 106
10.5.4 Preemption Configuration . . . 107
10.5.5 Preemption in Combination with License Orchestrator . . . 108
10.5.6 Common Use Cases . . . 108

11 Submitting Jobs from or to Windows hosts **110**
11.1 Job submission from a Windows submit host to a Windows execution host . . 111
11.1.1 Running Jobs in the foreground . . . 112
11.2 Job submission from an UNIX submit host to a Windows execution host . . . 113
11.3 Job submission from a Windows submit host to an UNIX execution host . . . 114

1 Overview of Basic User Tasks

Altair Grid Engine offers the following basic commands, tools and activities to accomplish common user tasks in the cluster:

Table 1: Basic tasks and their corresponding commands

Task	Command
submit jobs	qsub, qresub, qrsh, qlogin, qsh, qmake
check job status	qstat
modify jobs	qalter, qhold, qrls
delete jobs	qdel
check job accounting after job end	qacct
display cluster state	qstat, qhost, qselect, qquota
display cluster configuration	qconf

Note
`qsh` is not available on Microsoft Windows submit hosts and a `qsh` cannot be submitted to Windows execution hosts.

The next sections provide detailed descriptions of how to use these commands in a Altair Grid Engine cluster.

2 A Simple Workflow Example

Using Altair Grid Engine from the command line requires sourcing the settings file to set all necessary environment variables. The settings file is located in the `<AGE installation path>/<AGE cell>/common` directory. This directory contains two settings files for Unix: `settings.sh` for Bourne shell, bash and compatible shells, and `settings.csh` for `csh` and `tcsh`. If a Windows execution, submit or admin host is part of the Altair Grid Engine cluster, there is also a `settings.bat` for the Windows console (also known as `cmd.exe window`).

For simplicity, this document refers to the `<AGE installation path>` as `$SGE_ROOT` and the `<AGE_CELL>` as `$SGE_CELL`. Both environment variables are set when the settings file is sourced.

Source the settings file. Choose one of the following commands to execute based on the shell type in use.

Bourne shell/bash:

```
# . $SGE_ROOT/$SGE_CELL/common/settings.sh
```

csh/tcsh:

```
# source $SGE_ROOT/$SGE_CELL/common/settings.csh
```

Windows console:

```
> %SGE_ROOT%\%SGE_CELL%\common\settings.bat
```

Now that the shell is set up to work with Altair Grid Engine, it is possible to check which hosts are available in the cluster by running the qhost command.

Sample qhost output:

```
# qhost
HOSTNAME     ARCH         NCPU  LOAD  MEMTOT  MEMUSE  SWAPTO  SWAPUS
-------------------------------------------------------------------
global       -               -     -       -       -       -       -
kailua       lx-amd64        4  1.03    7.7G    2.2G    8.0G     0.0
halape       lx-x86          2  0.00  742.8M   93.9M  752.0M     0.0
kahuku       lx-amd64        2  0.01  745.8M  103.8M  953.0M     0.0
```

The sample qhost output above shows three hosts available, all of which run Linux (`lx-`), two in 64 bit (`amd64`), one in 32 bit mode (`x86`). One provides 4 CPUs; the other two just 2 CPUs. Two hosts are idle but have approximately 740 MB RAM available, while the third is loaded by 25% (LOAD divided by NCPU) and has 7.7 GB RAM in total.

This sample cluster has more than enough resources available to run a simple example batch job. Use the qsub command to submit a batch job. From the example job scripts in `$SGE_ROOT/examples/jobs`, submit `sleeper.sh`.

Note

The following example applies only to UNIX submit and execution hosts. How to submit the following job from or to a Windows host is explained in Submitting Jobs from or to Windows hosts.

```
# qsub $SGE_ROOT/examples/jobs/sleeper.sh
Your job 1 ("Sleeper") has been submitted
```

The qsub command sent the job to the Qmaster to determine which execution host is best suited to run the job. Follow the job's different stages with the qstat command:

- Immediately after submission, the job is in state *qw* (*queued*, *waiting*) in the pending job list.

`qstat` shows the submit time (when the job was submitted to the Qmaster from the `qsub` command on the submit host).

```
# qstat
job-ID  prior   name       user    state   submit/start at     queue  slots ja-task-ID
-----------------------------------------------------------------------------------
    1 0.00000 Sleeper    jondoe    qw    03/10/2011 19:58:35               1
```

Note
If running on a Windows execution host, the job name will be “cmd.exe”.

- A few seconds later, `qstat` shows the job in state *r* (*running*) and in the run queue *all.q* on host *kahuku*.

Since the job is running, qstat shows the start time (when the job was started on the execution host). A priority was automatically assigned to the job. Priority assignment is explained later in this document.

```
# qstat
job-ID  prior name    user   state  submit/start at     queue      slots ja-task-ID
-----------------------------------------------------------------------------------
1 0.55500 Sleeper    jondoe   r   03/10/2011 19:58:42 all.q@kahuku       1
```

Note
Between the states *qw* and *r*, the job may be in state *t* (*transferring*) for a short time or state *l* (*waiting for license*). Occasionally, these states can also be seen in the `qstat` output.

While a job is running, use the `qstat -j <job-ID>` command to display its status:

```
# qstat -j 1
==============================================================
job_number:                 1
exec_file:                  job_scripts/1
submission_time:            Thu Mar 11 19:58:35 2011
owner:                      jondoe
uid:                        1000
group:                      users
gid:                        100
sge_o_home:                 /home/jondoe
sge_o_log_name:             jondoe
sge_o_path:                 /gridengine/bin/lx-amd64:/usr/local/sbin:
                            /usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:.
sge_o_shell:                /bin/tcsh
sge_o_workdir:              /gridengine
sge_o_host:                 kailua
account:                    sge
hard resource_list:         hostname=kailua
mail_list:                  jondoe@kailua
notify:                     FALSE
job_name:                   Sleeper
jobshare:                   0
shell_list:                 NONE:/bin/sh
```

```
env_list:
job_args:                   3600
script_file:                /gridengine/examples/jobs/sleeper.sh
binding:                    NONE
usage    1:                 cpu=00:00:00, mem=0.00000 GBs, io=0.00003,
                            vmem=8.008M, maxvmem=8.008M
binding    1:               NONE
scheduling info:            (Collecting of scheduler job information
                            is turned off)
```

This simple sleeper job does nothing but sleep on the execution host. It doesn't need input, but it outputs two files in the home directory of the user who submitted the job: `Sleeper.o1` and `Sleeper.e1`. The `Sleeper.e1` file contains whatever the job printed to stderr, and it should be empty if the job ran successfully. The `Sleeper.o1` file contains what the job printed to stdout, for example:

```
Here I am. Sleeping now at: Thu Mar 10 20:01:10 CET 2011
Now it is: Thu Mar 10 20:02:10 CET 2011
```

Altair Grid Engine also keeps records of this job, as shown with the qacct command:

```
# qacct -j 1
==============================================================
qname        all.q
hostname     kailua
group        users
owner        jondoe
project      NONE
department   defaultdepartment
jobname      Sleeper
jobnumber    10
taskid       undefined
account      sge
priority     0
qsub_time    Thu Mar 10 19:58:35 2011
start_time   Thu Mar 10 19:58:42 2011
end_time     Thu Mar 10 19:59:43 2011
granted_pe   NONE
slots        1
failed       0
exit_status  0
ru_wallclock 61
ru_utime     0.070
ru_stime     0.050
ru_maxrss    1220
ru_ixrss     0
ru_ismrss    0
ru_idrss     0
```

```
ru_isrss     0
ru_minflt    2916
ru_majflt    0
ru_nswap     0
ru_inblock   0
ru_oublock   176
ru_msgsnd    0
ru_msgrcv    0
ru_nsignals  0
ru_nvcsw     91
ru_nivcsw    8
cpu          0.120
mem          0.001
io           0.000
iow          0.000
maxvmem      23.508M
arid         undefined
```

Refer to the **accounting(5)** man page for the meaning of all the fields output by the qacct command.

3 Displaying Altair Grid Engine Status Information

3.1 Cluster Overview

Several commands provide different perspectives on Altair Grid Engine cluster status information.

- `qhost` displays the status of Altair Grid Engine hosts, queues and jobs from the host perspective.
- `qstat` shows information about jobs, queues, and queue instances.
- `qconf` command, which is mainly used by the administrator for configuring the cluster, also shows the configuration of the cluster. Use it to understand why the cluster makes some decisions or is in a specific state.

3.2 Hosts and Queues

Altair Grid Engine monitoring and management centers around two main configuration object types: hosts and queues.

- A host represents a node in the cluster, physical or virtual. Each host has an associated host configuration object that defines the properties of that host. In addition, Altair Grid Engine has a global host configuration object that defines default values for all host properties. Any host that either does not have an associated host configuration object or has a host configuration object that does not set values for all host properties will inherit all or some property values from the global host configuration object.

- A queue is a set of global configuration properties that govern all instances of the queue. An instance of a queue on a specific host inherits its queue configuration properties from the queue. A queue instance may, however, explicitly override some or all of the queue configuration properties.
- Jobs are executed on a host within the context of a queue instance. Pending jobs wait in a global pending job list where they wait to be assigned by the scheduler to a queue instance. Altair Grid Engine provides the following commands to display the states of these objects or to configure them:
- `qhost` shows the cluster status from the execution host perspective.
- `qstat` shows the cluster status from the job or queue perspective.
- `qconf` displays the cluster configuration and allows administrators to change configurations.

qhost

The `qhost` command shows the cluster status from the execution host perspective.

```
# qhost
```

Calling just `qhost` by itself prints a table that lists the following information about the execution hosts:

- architectures
- number of cores
- current load
- total RAM
- currently used RAM
- total swap space
- currently used swap space

The line "global" appears there, representing the global host, a virtual configuration object that provides defaults for all attributes of the real hosts that are not filled by real data. It's listed here just for completeness.

```
# qhost -q -j
```

- Using the `-j` option, qhost lists all currently running jobs underneath the hosts on which they are running.
- Using the `-q` option, qhost displays all queues that have instances on a host, underneath the corresponding host.

Using both switches at once, it's possible to get a comprehensive overview over the cluster in a relatively compact output format. To prevent lengthy output in larger clusters, qhost provides several options to filter the output.

- Use the -h hostlist option to display only the information about the listed hosts.
- Use the -l attr=val,... option to specify more complex filters. See section Requestable Attributes for more details.

For example, the following command displays only hosts of a specific architecture:

```
# qhost -l arch=lx-amd64
```

- Use the -u user,... option to show only jobs from the specified users. This implies the -j option.
- Use the -F [attribute] option to list either all the resources an execution host provides or just the selected ones.

See the qhost(1) man page for a detailed description of all options.

qstat

To view the cluster from the queue or job perspective, use the qstat command.

- Without any option, the qstat command lists all jobs of the current user.
- The -ext option can be added to most options of qstat and causes more attributes to be printed.
- With the -u "*" option (**the asterisk must be enclosed in quotes!**), the jobs of all users are displayed. With -u <user,...> only the jobs of the specified users are listed.
- With the -g c option, the status of all cluster queues is displayed.
- The -j <job-ID> option prints information about the specified job of the current user. With a list of job-IDs or "*", this information is printed for the specified jobs or all jobs of the current user.
- The -j option without any job-ID prints information about all pending jobs of the current user.

  ```
  # qstat -f
  ```

- The -f option shows the full output of all queue instances with the jobs running in them. By default, just the jobs of the current user; add -u "*" to get all jobs listed for all users.

  ```
  # qstat -F
  ```

- The -F option shows all resources the queue instances provide.

The following are several options to filter queues:

- By name (`-q queue_list`)
- By any provided resource (`-l resource_list`)
- By queue state (`-qs {a|c|d|o|s|u|A|C|D|E|S}`)
- By parallel environments (`-pe pe_list`)
- Access permissions for specific users (`-U user_list`) and to filter out queue instances where no job of the current or specified user(s) is running.

Jobs can also be filtered.

- by state (`-s {p|r|s|z|S|N|P|hu|ho|hs|hd|hj|ha|h|P|N|S|a}`)
- by the job submitting user (`-u user_list`)

3.3 Requestable Resources

Each Altair Grid Engine configuration object (global, queue, host) has several resources whose values are either reported by loadsensors, reported by the OS or configured by a manager or an operator.

These are resources such as the execution host architecture, number of slots in the queue, current load of the host or configured complex variables. A job can request to be executed in an environment with specific resources. These requests can be hard or soft: a hard request denotes that a job can run only in an environment that provides at least the requested resource, while a soft request specifies that the job should be executed in an environment that fulfills all soft requests as much as possible.

In all commands, no matter if they are made for job submission or if they are made for listing the provided resources, the option to specify the requested resources is always `-l <resource>=<value>`. Each resource has a value of one of the following types:

- boolean
- integer
- float
- string
- regular expression string

For example, the following command submits a job that can run on hosts with Solaris on a 64-bit Sparc CPU:

```
# qsub -l arch=sol-sparc64 job
```

By default, this is a hard request. To specify it as a soft request, the command would change to the following:

```
# qsub -soft -l arch=sol-sparc64 job
```

The `-soft` option denotes that all following `-l resource=value` requests should be seen as soft requests. With `-hard` the requests can be switched back to hard requests. This can be switched as often as necessary, as shown in the following example:

```
# qsub -soft -l arch=sol-sparc64 -hard -l slots>4 -soft -l h_vmem>300M -hard -l num_cpus>2 job
```

Using wildcards in resource requests is also permitted.

```
# qsub -l arch="sol-*" job
```

This command requests the job to be scheduled on any Solaris host.

Note

The quotes (`"`) are necessary to prevent the shell from expanding the asterisk `"*"`.

To show the list of resources a queue instance provides, enter the following command:

```
# qstat -F
```

Sample qstat output is shown below.

```
queuename                      qtype resv/used/tot. load_avg arch      states
---------------------------------------------------------------------------------
all.q@kailua                   BIPC  0/0/40         1.14     lx-amd64
    hl:arch=lx-amd64
    hl:num_proc=4
    hl:mem_total=7.683G
    hl:swap_total=7.996G
    hl:virtual_total=15.679G
    hl:load_avg=1.140000
    hl:load_short=1.150000
    hl:load_medium=1.140000
    hl:load_long=1.310000
    hl:mem_free=2.649G
    hl:swap_free=7.996G
    hl:virtual_free=10.645G
    hl:mem_used=5.034G
    hl:swap_used=0.000
    hl:virtual_used=5.034G
    hl:cpu=17.100000
```

```
hl:m_topology=SCTTCTT
hl:m_topology_inuse=SCTTCTT
hl:m_socket=1
hl:m_core=2
hl:m_thread=4
hl:np_load_avg=0.285000
hl:np_load_short=0.287500
hl:np_load_medium=0.285000
hl:np_load_long=0.327500
qf:qname=all.q
qf:hostname=kailua
qc:slots=40
qf:tmpdir=/tmp
qf:seq_no=0
qf:rerun=0.000000
qf:calendar=NONE
qf:s_rt=infinity
qf:h_rt=infinity
qf:s_cpu=infinity
qf:h_cpu=infinity
qf:s_fsize=infinity
qf:h_fsize=infinity
qf:s_data=infinity
qf:h_data=infinity
qf:s_stack=infinity
qf:h_stack=infinity
qf:s_core=infinity
qf:h_core=infinity
qf:s_rss=infinity
qf:h_rss=infinity
qf:s_vmem=infinity
qf:h_vmem=infinity
qf:min_cpu_interval=00:05:00
```

The resource list consists of three fields: `<type>:<name>=<value>`. The type is composed of two letters.

- The first letter denotes the origin of this resource.
  - `h` for host
  - `q` for queue
- The second letter denotes how the value is acquired.
  - `l` for load sensor
  - `f` for fixed, i.e. statically configured in the cluster, host or queue configuration
  - `c` for constant

3.4 User Access Permissions and Affiliations

In Altair Grid Engine, there are three general categories of users:

Table 2: User Categories

User Category	Description
managers	By default, there is always one default manager, the Altair Grid Engine administrator. Managers have universal permission in Altair Grid Engine.
operators	Operators have the permissions to modify the state of specific objects, e.g. enable or disable a queue.
other users	All other users only have permission to submit jobs, to modify and delete their own jobs, and to get information about the cluster status.

Managers are defined by the global manager list, which can be accessed through qconf options:

Table 3: `qconf` Options for Updating the Global Manager List

Option	Description
`-am user_list`	add user(s) to the manager list
`-dm user_list`	delete user(s) from the manager list
`-sm`	show a list of all managers

`qconf` provides the similar options for operators:

Table 4: `qconf` Options for Updating the Operator List

Option	Description
`-ao user_list`	add user to the operator list
`-do user_list`	delete user from the operator list
`-so`	show a list of all operators

By default, all users known to the operating system can use Altair Grid Engine as normal users. On Windows hosts, all normal Windows Active Domain users can use Altair Grid Engine as normal users if the short names are the same as on the UNIX hosts. Whenever a user name is used or configured in Altair Grid Engine, use the short name of the Windows Active Domain user name.

Each object of Altair Grid Engine uses the configuration values set in `user_list` and `xuser_list` to determine who is allowed to use an object. The `user_list` explicitly allows access, whereas the `xuser_list` explicitly disallows access. This access is controlled through corresponding, but opposite, values. For example, the lists have values `acl` and `xacl` which function exactly opposite of each other. If a user is disallowed in the global cluster configuration (by using `xacl`), he may not use any object of Altair Grid Engine: he may not submit any job, but he can still get information from the cluster using `qstat`, `qhost` and so on.

Users mentioned in the `user_list` are allowed to use Grid Engine, but users mentioned in the `xuser_list` are disallowed. If a user is mentioned in both, the `xuser_list` takes precedence, so he is disallowed to use the object. If a `user_list` is defined, only users mentioned there are allowed to use the object. If a `xuser_list` is defined and the `user_list` is undefined, then all users except the ones mentioned in the `xuser_list` are allowed to use the object.

Note
The `user_list` and `xuser_list` accept only user sets, not user names. So it's necessary to define user sets before using these options of qconf.

Table 5: qconf Options for Updating the User List

Option	Description
-au user_list listname_list	add user(s) to user set list(s)
-Au fname	add user set from file
-du user_list listname_list	delete user(s) from user set list(s)
-dul listname_list	delete user set list(s) completely
-mu listname_list	modify the given user set list
-Mu fname	modify user set from file
-su listname_list	show the given user set list
-sul	show a list of all user set lists

A user set contains more information than just the names of the users in this set: see the man page `access_list(5)` for details. User sets can be defined by specifying UNIX users and primary UNIX groups, which must be prefixed by an `@` sign. There are two types of user sets: Access lists (type `ACL`) and departments (type `DEPT`). Pure access lists allow enlisting any user or group in any access list.

When using departments, each user or group may only be enlisted in one department, in order to ensure a unique assignment of jobs to departments. For the jobs whose users do not match with any of the users or groups enlisted under entries, the default department is assigned.

Table 6: Man Pages to See for Further Reference

Subject	Man Pages
`user_list` and `xuser_list`	sge_conf(5), queue_conf(5), host_conf(5) and sge_pe(5)
`acl` and `xacl` lists	project(5)
user lists format	access_list(5)
options to specify users and user sets	qconf(1)

4 Submitting Batch Jobs

4.1 What is a Batch Job?

A batch job is a single, serial work package that gets executed without user interaction. This work package can be any executable or script that can be executed on the execution host. Attached to this work package are several additional attributes that define how Altair Grid Engine handles the job and that influence the behavior of the job.

4.2 How to Submit a Batch Job

From the command line, batch jobs are submitted using the qsub command. Batch jobs can also be submitted using the deprecated GUI qmon or using the DRMAA (Distributed Resource Management Application) interface.

Note

qmon and DRMAA are not supported on Windows submit hosts.

Batch jobs are typically defined by a script file located at the submit host. This script prepares several settings and starts the application that does the real work. Altair Grid Engine transfers this script to the execution host, where it gets executed. Alternately, the script can be read from stdin instead of from a file. For a job that is just a binary to be executed on the remote host, the binary is typically already installed on the execution host, and therefore does not need to be transferred from the submit host to the execution host.

Note

The default shell for a queue is /bin/sh. As the Grid Engine Administrator you can change the default shell by modifying the shell parameter in the queue configuration (qconf -mq <queue-name>).

4.2.1 Example 1: A Simple Batch Job

To submit a simple batch job that uses a job script and default attributes, run the following command:

```
# qsub $SGE_ROOT/examples/jobs/simple.sh
```

Note

See Windows examples for how to submit the following examples jobs from or to a Windows host.

If this command succeeds, the `qsub` command should print the following note:

```
Your job 1 ("simple.sh") has been submitted
```

Now check the status of the job while the job is running:

```
# qstat
```

If `qstat` does not print any information about this job, it has already finished. Note that `simple.sh` is a short running job. The output of the job will be written to `~/simple.sh.o1` and the error messages to `~/simple.sh.e1`, where ~ is the home directory on the execution host of the user who submitted the job.

4.2.2 Example 2: An Advanced Batch Job

`qsub` allows several attributes and requirements to be defined using command line options at the time the job is submitted. These attributes and requirements can affect how the job gets handled by Altair Grid Engine and how the job script or binary is executed. For example, the following command defines these attributes of the job:

```
# qsub -cwd -S /bin/xyshell -i /data/example.in -o /results/example.out -j
  y example.sh arg1 arg2
```

Table 7: Explanation of Command Line Options in Example 2

Option	Description
-cwd	The job will be executed in the same directory as the current directory
-S /bin/xyshell	The shell /bin/xyshell will be used to interpret the job script.
-i /data/example.in	The file "/data/example.in" on the execution host will be used as input file for the job.
-o /results/example.out	The file "/results/example.out" on the execution host will be used as output file for the job.
-j y	Job output to stderr will be merged into the "/results/example.out" file.
example.sh arg1 arg2	The job script is "example.sh" must exist locally and gets transferred to the execution host by Altair Grid Engine. arg1 and arg2 will be passed to this job script.

4.2.3 Example 3: Another Advanced Batch Job

```
# qsub -N example3 -P testproject -p -27 -l a=lx-amd64 example.sh
```

Table 8: Explanation of Command Line Options in Example 3

Option	Description
-N example2	The job will get the name "example3" instead of the default name which is the name of the job script.
-P testproject	The job will be part of the project "testproject".
-p -27	The job will be scheduled with a lower priority than by default.
-l a=lx-amd64	The job can get scheduled only to a execution host that provides the architecture "lx-amd64".
example.sh	The job script without any arguments.

4.2.4 Example 4: A Simple Binary Job

```
# qsub -b y firefox
```

The `-b y` option tells Altair Grid Engine that this is a binary job; the binary does already exist on the execution host and doesn't have to be transferred by Altair Grid Engine from the submit to the execution host.

See the `qsub(5)` man page for an explanation of all possible qsub options.

4.3 Specifying Requirements

`qsub` provides three options to specify the requirements that must be fulfilled in order to run the job on the execution host. These are requirements like the host architecture, available memory, required licenses, specific script interpreters installed, and so on.

These resource requirements are specified on the qsub command line using the `-l` option. For example, to ensure the job gets scheduled only to a host that provides the architecture type `lx-x86`, i.e. Linux on a x86 compatible 32 bit CPU, issue the following qsub option:

```
# qsub -l arch=lx-x86 my_job.sh
```

Specifying several requirements at once and using wildcards inside a requirement is possible, as in the following example:

```
# qsub -l a="sol-*|*-amd" -l h="node??" job.sh
```

This example specifies that the job requests must be scheduled to a host whose architecture string starts with `sol-` and/or ends with `amd64`. At the same time, the hostname of the execution host must start with `node` and have exactly two additional trailing characters.

There are two different kinds of requests, hard and soft requests.

- A hard request must be fulfilled in order to schedule the job to the host.
- A soft request should be fulfilled. Grid Engine tries to fulfill as many soft requests as possible.

By default, all requests specified by the `-l` option are hard requests. The `-soft` option switches the behaviour: starting with the `-soft` option, all subsequent requests are considered soft requests. A “-hard” option in the command line switches back to hard requests. “-hard” and “-soft” can be specified as often as necessary.

Example:

```
# qsub -soft -l host="node??" -hard -l h_vmem=2G -l arch="sol*" -soft -l cpu=4
```

As described above in the section Requestable Resources, the attributes that are provided by all queue instances can be listed using qstat:

```
# qstat -F
```

To specify a particular queue instance, use the -q option:

```
# qstat -F -q all.q@kailua
```

As an alternative to specifying job requirements on the command line each time a job is submitted, default requirements can be specified by the job submitting user and the Altair Grid Engine administrator.

Requirements are evaluated in the following order:

- Request files
- Requests in job script
- Command line
- Options defined later (e.g., at command line) override options defined earlier (e.g., in the job script)

Note
Note that soft and hard requirements are collected separately.

4.3.1 Request Files

Request files allow options to be set automatically for all jobs submitted. Request files are read in the following order:

- The global request file $SGE_ROOT/$SGE_CELL/default/sge_request
- The private user request file $HOME/.sge_request
- The application specific request file $cwd/.sge_request

- The qsub command line

Since the request files are read in order, any option defined in more than one of them is overridden by the last-read occurrence, except for options that can be used multiple times on a command line. The resulting options are used as if they were written in the qsub command line, while the real qsub command line is appended to it, again overriding options that were specified in one of the three files. At any time, the "-clear" option can be used to discard all options that were defined previously.

In these request files, each line can contain one or more options in the same format as in the qsub command line. Lines starting with the hash sign (#) in the first column are ignored. See the sge_request(5) man page for additional information.

4.3.2 Requests in the Job Script

Note
Specifying requests in a Windows job script is not supported.

Submit options can also be defined in the jobs script. Each line of the job script that starts with `#$` or with the prefix that is defined using the `-C` option is considered to be a line that contains submit options, as in the following example:

```
#!/bin/sh

#$ -P testproject
#$ -o test.out -e test.err

echo "Just a test"
```

These options are read and parsed before the job is submitted and are added to the job object. The location where in the job script these options are defined does not matter, but the order matters - if two options override each other, the last one wins.

5 Using Job Classes to Prepare Templates for Jobs

When Altair Grid Engine jobs are submitted then various submit parameters have to be specified either as switches which are passed to command line applications or through corresponding selections in the graphical user interface. Some of those switches define the essential characteristics of the job, others describe the execution context that is required so that the job can be executed successfully. Another subset of switches needs to be specified only to give Altair Grid Engine the necessary hints on how to handle a job correctly so that it gets passed through the system quickly without interfering with other jobs.

In small and medium sized clusters with a limited number of different job types this is not problematic. The number of arguments that have to be specified can either be written into

default request files, embedded into the job script, put into an option file (passed with -@ of qsub) or they can directly be passed at the command line.

Within larger clusters or when many different classes of jobs should run in the cluster then the situation is more complex and it can be challenging for a user to select the right combination of switches with appropriate values. Cluster managers need to be aware of the details of the different job types that should coexist in the cluster so that they can setup suitable policies in line with the operational goals of the site. They need to instruct the users about the details of the cluster setup so that these users are able to specify the required submission requests for each job they submit.

Job classes have been introduced in Altair Grid Engine 8.1 to be able to:

- Specify job templates that can be used to create new jobs.
- Reduce the learning curve for users submitting jobs.
- Avoid errors during the job submission or jobs which may not fit site requirements.
- Ease the cluster management for system administrators.
- Provide more control to the administrator for ensuring jobs are in line with the cluster set-up.
- Define defaults for all jobs that are submitted into a cluster.
- Improve the performance of the scheduler component and thereby the throughput in the cluster.

5.1 Examples Motivating the Use of Job Classes

Imagine you have users who often make mistakes specifying memory limits for a specific application called memeater. You want to make it easy for them by specifying meaningful defaults but you also want to give them the freedom to modify the memory limit default according to their needs. Then you could use the following job class configuration (only an excerpt of the full configuration is shown):

```
jcname          memeater
variant_list    default
owner           NONE
user_lists      NONE
xuser_lists     NONE
...
CMDNAME         /usr/local/bin/memeater
...
l_hard          {~}{~}h_vmem=6GB
...
```

Without going into the specifics of the job class syntax, the above job class will use a default of 6 GB for the memory limit of the job. It will however be feasible for users to modify this limit. Here are two examples for how users would submit a job based on this job class. The

first maintaining the default, the second modifying it to 8 GB (again without going into the details of the syntax being used here):

```
qsub -jc memeater
qsub -jc memeater -l h_vmem=8GB
```

Now assume a slightly modified scenario where you want to restrict a certain group of users called novice to only use the preset of 6 GB while another group of users called expert can either use the default or can modify the memory limit. The following job class example would accomplish this. And the trick is that job classes support so called variants as well as user access lists:

```
jcname          memeater
variant_list    default, advanced
owner           NONE
user_lists      novice, [advanced=expert]
xuser_lists     NONE
...
CMDNAME         /usr/local/bin/memeater
...
l_hard          h_vmem=6GB,[{~}advanced={~}h_vmem=6GB]
...
```

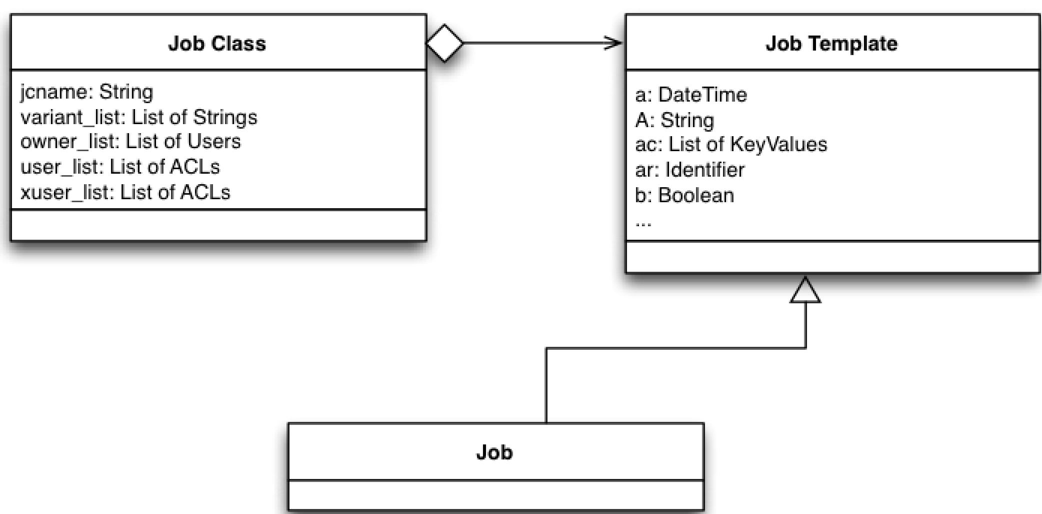
With this job class configuration, the novice users would only be able to submit their job using the first command example below while expert users could use both examples:

```
qsub -jc memeater
qsub -jc memeater.advanced -l h_vmem=8GB
```

The two use cases for job classes above are only snippets for all the different scenarios to which job classes may be applied and they only provide a glimpse onto the features of job classes. The next sections describe all attributes forming a job class object, commands that are used to define job classes as well as how these objects are used during job submission to form new jobs. A set of examples with growing functionality will illustrate further use cases. This will be followed by describing how job classes can be embedded with other parts of a Altair Grid Engine configuration to extract the maximum benefit from job classes. Finally, specific means for monitoring job class jobs will be shown.

5.2 Defining Job Classes

A job class is a new object type in Altair Grid Engine. Objects of this type can be defined by managers and also by users of a Altair Grid Engine Cluster to prepare templates for jobs. Those objects can later on be used to create jobs.



Like other configuration objects in Altair Grid Engine each job class is defined by a set of configuration attributes. This set of attributes can be divided into two categories. The first category contains attributes defining a job class itself and the second category all those which form the template which in turn eventually gets instantiated into new jobs.

5.2.1 Attributes describing a Job Class

Following attributes describe characteristics of a job class:

Table 9: Job Class Attributes

Attribute	Value specification
jcname	The jcname attribute defines a name that uniquely identifies a job class. Please note that NO_JC and ANY_JC are reserved keywords that cannot be used as names for new job classes. There is one particular job class with the special name template. It acts as template for all other job classes and the configuration of this job class template can only be adjusted by users having the manager role in Altair Grid Engine. This gives manager accounts control about default settings, some of which also can be set so that they must not be changed (see below for more information on how to enforce options).
variant_list	Job classes may, for instance, represent an application type in a cluster. If the same application should be started with various different settings in one cluster or if the possible resource selection applied by Altair Grid Engine system should depend on the mode how the application should be executed then it is possible to define one job class with multiple variants. A job class variant can be seen as a copy of a job class that differs only in some aspects from the original job class.

Attribute	Value specification
	The variant_list job class attribute defines the names of all existing Job Class variants. If the keyword NONE is used or when the list contains only the word default then the job class has only one variant. If multiple names are listed here, that are separated by commas, then the job class will have multiple variants. The default variant always has to exist. If the variant_list attribute does not contain the word default then it will be automatically added by the Altair Grid Engine system. Other commands that require a reference of a job class can either use the jcname to refer to the default variant of a job class or they can reference a different variant by combining the jcname with the name of a specific variant. Both names have to be separated by a dot (.) character.
owner_list	The owner_list attribute denotes the ownership of a job class. As default the user that creates a job class will be the owner. Only this user and all managers are allowed to modify or delete the job class object. Managers and owners can also add additional user names to this list to give these users modify and delete permissions. If a manager creates a job class then the owner_list will be NONE to express that only managers are allowed to modify or delete the corresponding job class. Even if a job class is owned only by managers it can still be used to create new jobs. The right to derive new jobs from a job class can be restricted with the user_list and xuser_list attributes explained below.
user_list	The user_list job class parameter contains a comma separated list of Altair Grid Engine user access list names or user names. User names have to be prefixed with a percent character (%). Each user referenced in the user_list and each user in at least one of the enlisted access lists has the right to derive new jobs from this job class using the -jc switch of one of the submit commands. If the user_list parameter is set to NONE (the default) any user can use the job class to create new jobs if access is not explicitly excluded via the xuser_lists parameter described below. If a user is contained both in an access list enlisted in xuser_lists and user_lists the user is denied access to use the job class.
xuser_list	The xuser_list job class contains a comma separated list of Altair Grid Engine user access list names or user names. User names have to be prefixed with a percent character (%). Each user referenced in the xuser_list and each user in at least one of the enlisted access lists is not allowed to derive new jobs from this job class. If the xuser_list parameter is set to NONE (the default) any user has access. If a user is contained both in an access list enlisted in xuser_lists and user_lists the user is denied access to use the job class.

5.2.2 Example 1: Job Classes - Identity, Ownership, Access

Below you can find an example for the first part of a sleeper job class. It will be enhanced in each of the following chapters to illustrate the use of job classes.

```
jcname        sleeper
variant_list  NONE
owner         NONE
user_lists    NONE
xuser_lists   NONE
...
```

`sleeper` is the unique name that identifies the job class (`jcname sleeper`). This job class defines only the default variant because no other variant names are specified (`variant_list NONE`). The job class does not specify an owner (`owner NONE`) as a result it can only be changed or deleted by users having the manager role. Managers and all other users are allowed to derive new jobs from this job class. Creating new jobs is not restricted (`user_lists NONE`; `user_lists NONE`).

5.2.3 Attributes to Form a Job Template

Additionally to the attributes mentioned previously each job class has a set of attributes that form a job template. In most cases the names of those additional attributes correspond to the names of command line switches of the qsub command. The value for all these additional attributes might either be the keyword `UNSPECIFIED` or it might be the same value that would be passed with the corresponding qsub command line switch.

All these additional job template attributes will be evaluated to form a virtual command line when a job class is used to instantiate a new job. All attributes for which the corresponding value contains the `UNSPECIFIED` keyword will be ignored whereas all others define the submit arguments for the new job that will be created.

All template attributes can be divided in two groups. There are template attributes that accept simple attribute values (like a character sequence, a number or the value yes or no) and there are template attributes that allow to specify a list of values or a list of key/value pairs, like the list of resource requests a job has or the list of queues where a job might get executed.

The table below contains all available template attributes. The asterisk character (`*`) tags all attributes that are list based. Within the description the default for each attribute is documented that will be used when the keyword `UNSPECIFIED` is used in the job class definition.

Table 10: Job Class Attributes to Form a Job Template

Attribute	Value specification
a	Specifies the time and date when a job is eligible for execution. If unspecified the job will be immediately eligible for execution. Format of the character sequence is the same as for the argument that might be passed with `qsub -a`.
A	Account string. The string `sge` will be used when there is no account string specified or when it is later on removed from a job template or job specification.
ac *	List parameter defining the name/value pairs that are part of the job context. Default is an empty list.
ar	Advance reservation identifier used when jobs should be part of an advance reservation. As default no job will be part of an advance reservation.
b	yes or no to express if the command should be treated as binary or not. The default for this parameter is no, i.e. the job is treated as a script.
binding	Specifies all core binding specific settings that should be applied to a job during execution. Binding is disabled as default.
CMDARG *	Defines a list of command line arguments that will be passed to CMDNAME when the job is executed. As default this list is empty.
CMDNAME *	Specified either the job script or the command name when binary submission is enabled (b yes). Please note that script embedded flags within specified job scripts will be ignored.
c_interval	Defines the time interval when a checkpoint-able job should be checkpointed. The default value is 0.
c_occasion	Letter combination that defines the state transitions when a job should be triggered to write a checkpoint. Default is 'n' which will disable checkpointing.
ckpt	Checkpoint environment name which specifies how to checkpoint the job. No checkpoint object will be referenced as default.
cwd	Specifies the working directory for the job. Path aliasing will not be used when this value is specified in a job class. In case of absence the home directory of the submitting user will be used as directory where the job is executed.
dl	Specifies the deadline initiation time for a job (see the chapter about deadline urgency in the administrators guide for more information). As default jobs have do defined deadline.
e *	List parameter that defines the path for the error file for specific execution hosts. As default the file will be stored in the home directory of the submitting user and the filename will be the combination of the job name and the job id.
h	yes or no to indicate if a job should be initially in hold state. The default is no.
hold_jid *	List parameter to create initial job dependencies between new jobs and already existing ones. The default is an empty list.
hold_jid_ad *	List parameter to create initial array job dependencies between new array jobs and already existing ones. The default is an empty list.

Attribute	Value specification
i *	List parameter that defines the path for the input file for specific execution hosts.
j	yes or no to show if error and output stream of the job should be joined into one file. Default is no.
js	Defines the job share of a job relative to other jobs. The default is 0.
l_hard *	List parameter that defines hard resource requirements of a job in the form of name/value pairs. The default is an empty list.
l_soft *	List parameter defining soft requests of a job. The default is an empty list.
mbind	Specifies memory binding specific settings that should be applied to a job during execution. Memory binding is disabled as default.
m	Character sequence that defines the circumstances when mail that is related to the job should be send. The default is 'n' which means no mails should be send.
M *	list parameter defining the mail addresses that will be used to send job related mail. The default is an empty list.
masterq *	List parameter that defines the queues that might be used as master queues for parallel jobs. The default is an empty list.
N	Default name for jobs. For jobs specifying a job script which are submitted with qsub or the graphical user interface the default value will be the name of the job script. When the script is read from the stdin stream of the submit application then it will be STDIN. qsh and qlogin jobs will set the job name to INTERACTIVE. qrsh jobs will use the first characters of the command line up to the first occurrence of a semicolon or space character.
notify	yes or no to define if warning signals will be send to a jobs if it exceeds any limit. The default is no
now	yes or no to specify if created jobs should be immediate jobs. The default is no.
o *	List parameter that defines the path for the output file for specific execution hosts.
P	Specifies the project to which this job is assigned.
p	Priority value that defines the priority of jobs relative to other jobs. The default priority is 0.
pe_name	Specifies the name of the parallel environment that will be used for parallel jobs. PE name pattern are not allowed. As default there is no name specified and as a result the job is no parallel job.
pe_range	Range list specification that defines the amount of slots that are required to execute parallel jobs. This parameter must be specified when also the pe_name parameter is specified.
q_hard *	List of queues that can be used to execute the job. Queue name pattern are not allowed. The default is an empty list.
q_soft *	List of queues that are preferred to be used when the job should be executed. Queue name pattern are not allowed. The default is an empty list.
R	yes or no to indicate if a reservation for this job should be done. The default is no.

Attribute	Value specification
r	yes or no to identify if the job will be rerun-able. The default is no.
S *	List parameter that defines the path of the shell for specific execution hosts. The default is an empty list.
shell	yes or no to specify if a shell should be executed for binary jobs or if the binary job should be directly started. The default is yes
t	Defines the task ID range for array jobs. Jobs are no array jobs as default.
V	yes or no. yes causes that all environment variables active during the submission of a job will be exported into the environment of the job.
v *	List of environment variable names and values that will be exported into the environment of the job. If also V yes is specified then the variable values that are active during the submission might be overwritten.

5.2.4 Example 2: Job Classes - Job Template

Second version of the sleeper job class defining job template attributes for the default variant:

```
jcname          sleeper
variant_list    NONE
owner           NONE
user_lists      NONE
xuser_lists     NONE
A               UNSPECIFIED
a               UNSPECIFIED
ar              UNSPECIFIED
b               yes
binding         UNSPECIFIED
c_interval      UNSPECIFIED
c_occasion      UNSPECIFIED
CMDNAME         /bin/sleep
CMDARG          60
ckpt            UNSPECIFIED
ac              UNSPECIFIED
cwd             UNSPECIFIED
display         UNSPECIFIED
dl              UNSPECIFIED
e               UNSPECIFIED
h               UNSPECIFIED
hold_jid        UNSPECIFIED
i               UNSPECIFIED
j               UNSPECIFIED
js              UNSPECIFIED
l_hard          UNSPECIFIED
l_soft          UNSPECIFIED
```

```
m               UNSPECIFIED
M               UNSPECIFIED
masterq         UNSPECIFIED
mbind           UNSPECIFIED
N               Sleeper
notify          UNSPECIFIED
now             UNSPECIFIED
o               UNSPECIFIED
P               UNSPECIFIED
p               UNSPECIFIED
pe_name         UNSPECIFIED
q_hard          UNSPECIFIED
q_soft          UNSPECIFIED
R               UNSPECIFIED
r               UNSPECIFIED
S               /bin/sh
shell           UNSPECIFIED
V               UNSPECIFIED
v               UNSPECIFIED
```

Most of the job template attributes are UNSPECIFIED. As a result the corresponding attributes will be ignored and the defaults of the submit client will be used when new jobs are created. When a job is derived from this job class then it will create a job using binary submission (b yes) to start the script /bin/sleep (CMDNAME /bin/sleep). 60 will be passed as command line argument to this script (CMDARG 60). The name of the job that is created will be Sleeper (N Sleeper) and the shell /bin/sh will be used to start the command (S /bin/sh). The definition of the sleeper job class is complete. Now it can be used to submit new jobs:

```
> qsub -jc sleeper
Your job 4097 ("Sleeper") has been submitted


> qsub -S /bin/sh -N Sleeper -b y /bin/sleep
Your job 4098 ("Sleeper") has been submitted
```

Job 4097 is derived from a job class whereas job 4098 is submitted conventionally. The parameters specified in the sleeper job class are identical to the command line arguments that are passed to qsub command to submit the jobs. As a result both jobs are identical. Both use the same shell and job command and therefore they will sleep for 60 seconds after start. The only difference between the two jobs is the submit time and the job id. Users that try to change both jobs after they have been submitted will also encounter an additional differences. It is not allowed to change the specification of job 4097. The reason for this is explained in the next chapter.

5.2.5 Access Specifiers to Allow Deviation

Access specifiers are character sequences that can be added to certain places in job class specifications to allow/disallow operations that can be applied to jobs that are derived from

that job class. They allow you to express, for instance, that job options defined in the jobs class can be modified, deleted or augmented when submitting a job derived from a job class. This means the job class owner can control how the job class can be used by regular users being allowed to derive jobs from this job class. This makes using job classes simple for the end user (because of a restricted set of modifications). It also avoids errors as well as the need to utilize Job Submission Verifiers for checking on mandatory options.

By default, if no access specifiers are used, all values within job classes are fixed. This means that jobs that are derived from a job class cannot be changed. Any attempt to adjust a job during the submission or any try to change a job after it has been submitted (e.g. with qalter) will be rejected. Also managers are not allowed to change the specification of defined in a job class when submitting a job derived from the job class.

To soften this restriction, job class owners and users having the manager role in a job class can add access specifiers to the specification of a job class to allow deviation at certain places. Access specifiers might appear before each value of a job template attribute and before each entry in a list of key or key/value pairs. The preceding access specifier defines which operations are allowed with the value that follows.

The full syntax for a job class template attribute is defined as <jc_templ_attr>:

```
<jc_templ_attr> := <templ_attr> | <list_templ_attr>
<templ_attr> := <attr_name> " " <attr_access_specifier>(<attr_value>|"UNSPECIFIED")
<list_templ_attr> := <list_attr_name> " " <attr_access_specifier> <list_attr_value>
<list_attr_value> := <access_specifier> ( (<list_entry> [ "," <access_specifier>
                                          <list_entry>, ...] ) | "UNSPECIFIED" )
<attr_access_specifier> :=  <access_specifier>
```

Please note the distinction between <attr_access_specifier> and <access_specifier>. <attr_access_specifier> is also an <access_specifier> but it is the first one that appears in the definition of list based job template attributes and it is the reason why two access specifiers might appear one after another. The first access specifier regulates access to the list itself whereas the following ones define access rules for the entries in the list they are preceding. These access specifiers (<access_specifier>) are available:

Table 11: Available Access Specifiers

Access Specifier	Description
	The absence of an access specifier indicates that the corresponding template attribute (or sublist entry) is fixed. Any attempt to modify or delete a specified value or any attempt to add a value where the keyword UNSPECIFIED was used will be rejected. It is also not allowed to add additional entries to lists of list based attributes if a list is fixed.
{-}	Values that are tagged with the {-} access specifier are removable. If this access specifier is used within list based attributes then removal is only allowed if the list itself is also modifiable. If all list entries of a list are removable then also the list itself must be removable so that the operation will be successful.

Access Specifier	Description
{~}	Values that are prefixed with the {~} access specifier can be changed. If this access specifier is used within list based attributes then the list itself must also be modifiable.
{~-} or {-~}	The combination of the {-} and {~} access specifiers indicates that the value it precedes is modifiable and removable.
{+}UNSPECIFIED or {+...}	The {+} access specifier can only appear in combination with the keyword UNSPECIFIED or before list attribute values but not within access specifiers preceding list entries. If it appears before list attribute values it can also be combined with the {~} and {-} access specifiers. This access specifier indicates that something can be added to the specification of a job after it has been submitted. For list based attributes it allows that new list entries can be added to the list.

5.2.6 Example 3: Job Classes - Access Specifiers

Here follows the third refinement of the sleeper job class giving its users more flexibility:

```
jcname          sleeper
variant_list    NONE
owner           NONE
user_lists      NONE
xuser_lists     NONE
A               UNSPECIFIED
a               UNSPECIFIED
ar              UNSPECIFIED
b               yes
binding         UNSPECIFIED
c_interval      UNSPECIFIED
c_occasion      UNSPECIFIED
CMDNAME         /bin/sleep
CMDARG          60
ckpt            UNSPECIFIED
ac              UNSPECIFIED
cwd             UNSPECIFIED
display         UNSPECIFIED
dl              UNSPECIFIED
e               UNSPECIFIED
h               UNSPECIFIED
hold_jid        UNSPECIFIED
i               UNSPECIFIED
j               UNSPECIFIED
js              UNSPECIFIED
l_hard          {~+}{~}a=true,b=true,{-}c=true
l_soft          {+}UNSPECIFIED
m               UNSPECIFIED
```

```
M               UNSPECIFIED
masterq         UNSPECIFIED
mbind           UNSPECIFIED
N               {~-}Sleeper
notify          UNSPECIFIED
now             UNSPECIFIED
o               UNSPECIFIED
P               UNSPECIFIED
p               UNSPECIFIED
pe_name         UNSPECIFIED
q_hard          UNSPECIFIED
q_soft          UNSPECIFIED
R               UNSPECIFIED
r               UNSPECIFIED
S               /bin/sh
shell           UNSPECIFIED
V               UNSPECIFIED
v               UNSPECIFIED
```

Now it is allowed to modify or remove the name of sleeper jobs (N {~-}Sleeper). Users deriving jobs from this class are allowed to add soft resource requests (l_soft {+}UNSPECIFIED). New hard resource requests can be added and the ones which are specified within the job class can be adjusted (l_hard {~+}...) but there are additional restrictions: The access specifiers preceding the resource requests (l_hard ...{~}a=true,b=true,{-}c=true) allow the modification of the resource a, the deletion of the resource c whereas the value of resource b is fixed (no access specifier). Users that try to submit or modify jobs that would violate one of the access specifiers will receive an error message and the request is rejected.

Here are some examples for commands that will be successful:

```
> qsub -jc sleeper -N MySleeperName
> qsub -jc sleeper -soft -l new=true
> qsub -jc sleeper -l a=false,b=true,new=true
```

Here you can see some commands that will be rejected:

```
> qsub -jc sleeper /path/to/my_own_sleeper (CMDNAME is not modifiable)
> qsub -jc sleeper -l a=false,b=false,new=true (l_hard has requested resource b=true.
    This cannot be changed)
> qsub -jc sleeper -S /bin/tcsh (S job template attribute does not allow to modify
    the shell)
```

5.2.7 Different Variants of the same Job Class

Job classes represent an application type in a cluster. If the same application should be started with various different settings or if the possible resource selection applied by the

Altair Grid Engine system should depend on the mode how the application should be executed then it is possible to define one job class with multiple variants. So think of it as a way to use the same template for very similar types of jobs, yet with small variations. The variant_list job class attribute defines the names of all existing job class variants. If the keyword NONE is used or when the list contains only the word default then the job class has only one variant. If multiple names are listed here, separated by commas, then the job class will have multiple variants. The default variant always has to exist. If the variant_list attribute does not contain the word default then it will be automatically added by the Altair Grid Engine system upon creating the job class.

Attribute settings for the additional job class variants are specified similar to the attribute settings of queue instances or queue domains of cluster queues. The setting for a variant attribute has to be preceded by the variant name followed by an equal character ("=") and enclosed in brackets ("[" and "]").

The position where access specifiers have to appear is slightly different in this case. The next example will show this (see the l_soft and N attributes).

5.2.8 Example 4: Job Classes - Multiple Variants

The following example shows the excerpt of the sleeper job class with three different variants

```
jcname          sleeper
variant_list    default,short,long
owner           NONE
user_lists      NONE
xuser_lists     NONE
A               UNSPECIFIED
a               UNSPECIFIED
ar              UNSPECIFIED
b               yes
binding         UNSPECIFIED
c_interval      UNSPECIFIED
c_occasion      UNSPECIFIED
CMDNAME         /bin/sleep
CMDARG          60,[short=5],[long=3600]
ckpt            UNSPECIFIED
ac              UNSPECIFIED
cwd             UNSPECIFIED
display         UNSPECIFIED
dl              UNSPECIFIED
e               UNSPECIFIED
h               UNSPECIFIED
hold_jid        UNSPECIFIED
i               UNSPECIFIED
j               UNSPECIFIED
js              UNSPECIFIED
l_hard          {~+}{~}a=true,b=true,{-}c=true
```

```
l_soft          {+}UNSPECIFIED,[{~+}long={~}d=true]
m               UNSPECIFIED
M               UNSPECIFIED
masterq         UNSPECIFIED
mbind           UNSPECIFIED
N               {~-}Sleeper,[{~-}short=ShortSleeper],[long=LongSleeper]
notify          UNSPECIFIED
now             UNSPECIFIED
o               UNSPECIFIED
P               UNSPECIFIED
p               UNSPECIFIED
pe_name         UNSPECIFIED
q_hard          UNSPECIFIED
q_soft          UNSPECIFIED
R               UNSPECIFIED
r               UNSPECIFIED
S               /bin/sh
shell           UNSPECIFIED
V               UNSPECIFIED
v               UNSPECIFIED
```

The sleeper job class has now three different variants (variant_list default,short,long). To reference a specific job class variant the name of the job class has to be combined with the name of the variant. Both names have to be separated by a dot ("."). If the variant name is omitted then automatically the default variant is referenced.

```
> qsub -jc sleeper
Your job 4099 ("Sleeper") has been submitted

> qsub -jc sleeper.short
Your job 4100 ("ShortSleeper") has been submitted

> qsub -jc sleeper.long
Your job 4101 ("LongSleeper") has been submitted
```

The returned message from the submit commands already indicates that there are differences between the three jobs. The jobs have different names. Compared to the other jobs, the job 4101 has an additional soft resource request d=true (`l_soft ...,[{~+}long={~}d=true]`). Job 4100 that was derived from the `sleeper.short` job class variant has no soft requests. Nothing was explicitly specified here for this variant and therefore it will implicitly use the setting of the `sleeper.default` job class variant (`l_soft {+}UNSPECIFIED,...`). Moreover, the job name (see the N attribute) can be modified or removed for the default and short variant but is fixed for the long variant.

5.2.9 Enforcing Cluster Wide Requests with the Template Job Class

After a default installation of Altair Grid Engine 8.7.0 there exists one job class with the name `template`. This job class has a special meaning and it cannot be used to create new jobs. Its

configuration can only be adjusted by users having the manager role. This jobs class acts as parent job class for all other job classes that are created in the system.

The values of job template attributes in this template job class and the corresponding access specifiers restrict the allowed settings of all corresponding job template attributes of other job classes. As default the `{+}UNSPECIFIED` add access specifier and keyword is used in the template job class in combination with all job template attributes. Due to that any setting is allowed to other job class attributes after Altair Grid Engine 8.7.0 has been installed.

This parent-child relationship is especially useful when all jobs that are submitted into a cluster are derived from job classes. Managers might then change the settings within the template. All other existing job classes that violate the settings will then switch into the configuration conflict state. The owners of those job classes have to adjust the settings before new jobs can be derived from them. All those users that intend to create a new job class that violates the settings of the template job class will receive an error.

You will also want to use the template job class to enforce restrictions on the access specifiers which can be used in job classes. Since any job class, whether create by a manager account or by regular users, is derived from the template job class those derived job classes are bound to stay within the limits defined by the template job class. So parameters which have been defined as fixed in the template job class, for instance, cannot be modified in any job class created by a manager or user. Likewise, parameters which have a preset value but are configured to allow deletion only cannot be modified in derived job classes. The following table shows the allowed transitions:

Table 12: Allowed Access Specifier Transitions

Access Specifier in Template JC	Allowed Access Specifier in Child JC
…	…
UNSPECIFIED	UNSPECIFIED
{~}…	{~}…
	…
{-}…	{-}…
	{~}…
	UNSPECIFIED
	…
{-~}…	{-~}…
	{-}…
	{~}…
	UNSPECIFIED
	…
{+}…	{+}…
	{-~}…
	{-}…
	{~}…
	UNSPECIFIED
	…

5.3 Relationship Between Job Classes and Other Objects

To fully integrate job classes into the already existing Altair Grid Engine system the possibility is provided to create new relations between current object types (like queues, resource quotas, JSV) and job classes.

5.3.1 Resources Available for Job Classes

The profile of a job is defined by the resource requirements and other job attributes. Queues and host objects define possible execution environments where jobs can be executed. When a job is eligible for execution then the scheduler component of the Altair Grid Engine system tries to find the execution environment that fits best according to all job specific attributes and the configured policies so that this job can be executed.

This decision making process can be difficult and time consuming especially when certain jobs having special resource requirements should only be allowed to run in a subset of the available execution environments. The use of job classes might help here because job classes will give the scheduler additional information on which execution environments will or will not fit for a job. The need to evaluate all the details about available resources of an execution environment and about the job's requirements will be reduced or can be completely eliminated during the decision making process.

This is achieved by an additional parameter in the queue configuration which provides a direct association between queues and one or multiple job classes. This parameter is called `jc_list` and might be set to the value NONE or a list of job classes or job class variant names. If a list of names is specified then the special keyword `ANY_JC` and/or `NO_JC` might be used within the list to filter all those jobs that are in principle allowed to run in this queues. The following combinations are useful:

Value Description

NONE No job may enter the queue.

ANY_JC Jobs may enter the queue that were derived from a job class.

NO_JC Only jobs may enter the queue that were not derived from a job class.

ANY_JC, NO_JC Any job, independent if it was derived from a job class or not, may be executed in the queue. This is the default for any queue that is created in a cluster.

<list of JC names> Only those jobs may get scheduled in the queue if they were derived from one of the enlisted job classes. NO_JC,
<list of JC names> Only those jobs that were not derived from a job class or those that were derived from one of the enlisted job classes can be executed here. ———————————————————————————————————————-

: Useful Values for the jc_list Attribute of a Queue

This relationship helps the scheduler during the decision making to eliminate queues early without the need to further look at all the details like resource requirements. Managers of Grid Engine Clusters may want to take care that there is at least one queue in the cluster

available that use the ANY_JC keyword. Otherwise jobs of users who have defined their own job class will not get cluster resources. Also at least one queue using the NO_JC keyword may need to be available. Otherwise conventionally submitted jobs will not get scheduled.

5.3.2 Defining Job Class Limits

Resource quota sets can be defined to influence the resource selection in the scheduler. The jcs filter within a resource quota rule may contain a comma separated list of job class names. This parameter filters for jobs requesting a job class in the list. Any job class not in the list will not be considered for the resource quota rule. If no jcs filter is used, all job classes and jobs with no job class specification match the rule. To exclude a job class from the rule, the name can be prefixed with the exclamation mark (!). '!*' means only jobs with no job class specification.

Example: Resource Quota Set Using a Job Class Filter

```
`name max_virtual_free_on_lx_hosts_for_app_1_2`

`description "quota for virtual_free restriction"`

`enabled true`

`limit users {user1,user2} hosts {@lx_host} jcs {app1, app2} to vf=6G`

`limit users {*} hosts {@lx_host} jcs {other_app, !*} to vf=4G`
```

The example above restricts user1 and user2 to 6G virtual_free memory for all jobs derived from of job class app1 or app2 on each Linux host part of the @lx_hosts host group. All users that either do not derive from a job class or request the job class named other_app will have a limit of 4G.

5.3.3 JSV and Job Class Interaction

During the submission of a job multiple Job Submission Verifiers can be involved that verify and possibly correct or reject a job. With conventional job submission (without job classes) each JSV will see the job specification of a job that was specified at the command line via switches and passed parameters or it will see the job parameters that were chosen within the dialog of the GUI.

When Jobs are derived from a job class then the process of evaluation via JSV scripts is the same but the job parameters that are visible in client JSVs are different. A client JSV will only see the requested job class via a parameter named jc and it will see all those parameters that were specified at the command line. All parameters that are defined in the job class itself cannot be seen.

Job classes will be resolved within the sge_qmaster process as soon as a request is received that tries to submit a job that should be derived from a job class. The following steps are taken (simplified process):

```
1) Create a new job structure

2) Fill job structure with defaults values

3) Fill job structure with values defined in the job class
(This might overwrite default values)

4) Fill job structure with values defined at the command line
(This might overwrite default values and values that were defined in the job class)

5) Trigger server JSV to verify and possibly adjust the job
(This might overwrite default values, JC values and values specified at the command line)

6) Check if the job structure violates access specifiers
```

If the server JSV changes the jc parameter of the job in step 5 then the submission process restarts from step 1 using the new job class for step 3.

Please note that the violation of the access specifiers is checked in the last step. As result a server JSV is also not allowed to apply modifications to the job that would violate any access specifiers defined in the job class specification.

5.4 Commands to Adjust Job Classes

5.4.1 Creating, Modifying and Deleting Job Classes

Job Classes can be created, modified or deleted with the following commands.

- `qconf -ajc <jcname>`

This is the command to add a new job class object. It opens an editor and shows the default parameters for a job class. After changing, saving necessary values and closing the editor, a new job class is created.

- `qconf -Ajc <filename>`

Adds a new job class object with its specification being stored in the specified file.

- `qconf -djc <jcname>`

Deletes a job class object with the given name.

- `qconf -mjc <jcname>`

Opens an editor and shows the current specification of the job class with the name `<jcname>`. After changing attributes, saving the modifications and closing the editor, the object is modified accordingly.

Note
The qconf commands that open an editor are not supported on Windows hosts. Instead, redirect the output of the corresponding `qconf -s...` command to a file, edit it there and apply the changes using `qconf -M...`, or simply use a UNIX host.

- `qconf -Mjc <filename>`

Modifies a job class object from file.

- `qconf -sjc <jcname>`

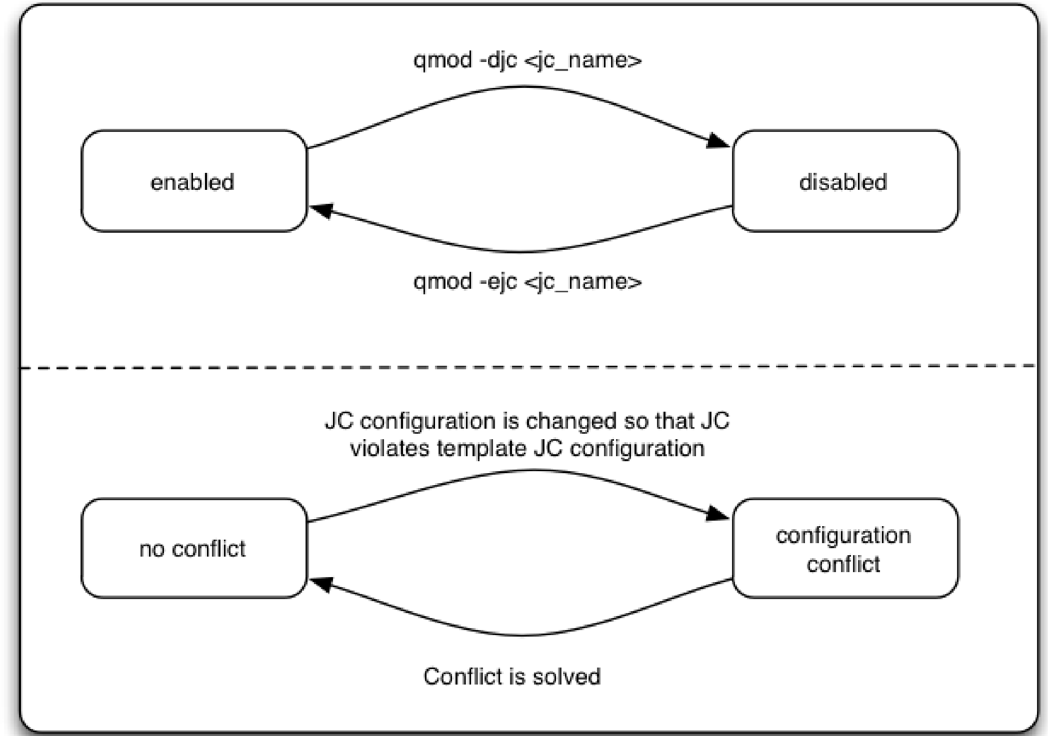
Shows the current specification of the job class with the name `<jcname>`.

- `qconf -sjcl`

Shows all names of existing job class objects that exist in a cluster.

5.4.2 States of Job Classes

Job Classes have a combined state that is the result of following the sub states: *enabled/disabled*, *no conflict/configuration conflict*



The *enabled/disabled* state is a manual state. A state change from enabled to disabled can be triggered with the `qmod -djc <jcname>` command. The command `qmod -ejc <jcname>` command can be used to trigger a state change from disabled to enabled. Job Classes in the disabled state cannot be used to create new jobs.

The *no conflict/configuration conflict* state is an automatic state that cannot be changed manually. Job classes that do not violate the configuration of the `template` job class are in the *no conflict* state. A job class in this state can be used to create new jobs (if it is also in enabled state). If the `template` job class or a derived job class is changed so that either a configuration setting or one of the access specifiers of the `template` job class is violated then the derived job class will automatically switch from the `no conflict` into the `configuration conflict` state. This state will also be left automatically when the violation is eliminated.

5.5 Using Job Classes to Submit New Jobs

Job Classes that are in the enabled and *no conflict* state can be used to create new jobs. To do this a user has to pass the `-jc` switch in combination with the name of a job class to a submit command like `qsub`. If the user has access to this job class then a new job will be created and all job template attributes that are defined in the job class will be used to initialize the corresponding parameters in the submitted job.

Depending on the access specifiers that are used in the job class it might be allowed to adjust certain parameters during the submission of the job. In this case additional switches and parameters might be passed to the submit command. All these additionally passed parameters will be used to adjust job parameters that where derived from the job class.

Additionally to the typical switches that are used to define job parameters there is a set of switches available that allow to remove parameters or to adjust parts of list based parameters in a job specification. The same set of switches can also be used with the modification command `qalter` to adjust job parameters after a job has already been created.

- `qsub/qalter -clearp <attr_name>`

The -clearp switch allows to remove a job parameter from the specification of a job as if it was never specified. What this means depends on the job parameter that is specified by <attr_name>. For all those attributes that would normally have a default value this default value will be set for all others the corresponding attribute will be empty. Parameter names that can be specified for <attr_name> are all the ones that are specified in the table above showing job template attribute names.

- `qsub/qalter -clears <list_attr_name> <key>`

This switch allows to remove a list entry in a list based attribute of a job specification. <list_attr_name> might be any name of a job template attribute that is tagged with the asterisk (*) in the table above. <key> has to be the name of the key of the sublist entry for key/value pairs or the value itself that should be removed when the list contains only values

- `qsub/qalter -adds <list_attr_name> <key> <value>`

-adds adds a new entry to a list based parameter.

- `qsub/qalter -mods <list_attr_name> <key> <value>`

The -mods switch allows to modify the value of a key/value pair within a list based job parameter.

5.6 Example: Submit a Job Class Job and Adjust Some Parameters

Assume that the following job class is defined in you cluster:

```
jcname          sleeper
variant_list    default,short,long
owner           NONE
user_lists      NONE
xuser_lists     NONE
A               UNSPECIFIED
a               UNSPECIFIED
ar              UNSPECIFIED
b               yes
binding         UNSPECIFIED
c_interval      UNSPECIFIED
c_occasion      UNSPECIFIED
CMDNAME         /bin/sleep
CMDARG          60,[short=5],[long=3600]
ckpt            UNSPECIFIED
ac              UNSPECIFIED
cwd             UNSPECIFIED
display         UNSPECIFIED
dl              UNSPECIFIED
e               UNSPECIFIED
h               UNSPECIFIED
hold_jid        UNSPECIFIED
i               UNSPECIFIED
j               UNSPECIFIED
js              UNSPECIFIED
l_hard          {~+}{~}a=true,b=true,{-}c=true
l_soft          {+}UNSPECIFIED,[{~+}long={~}d=true]
m               UNSPECIFIED
M               UNSPECIFIED
masterq         UNSPECIFIED
mbind           UNSPECIFIED
N               {~-}Sleeper,[{~-}short=ShortSleeper],[{~-}long=LongSleeper]
notify          UNSPECIFIED
now             UNSPECIFIED
o               UNSPECIFIED
P               UNSPECIFIED
p               UNSPECIFIED
```

```
pe_name         UNSPECIFIED
q_hard          UNSPECIFIED
q_soft          UNSPECIFIED
R               UNSPECIFIED
r               UNSPECIFIED
S               /bin/sh
shell           UNSPECIFIED
V               UNSPECIFIED
v               UNSPECIFIED
```

Now it is possible to submit jobs and to adjust the parameters of those jobs during the submission to fit specific needs:

```
1) qsub -jc sleeper -N MySleeper
2) qsub -jc sleeper.short -clearp N
3) qsub -jc sleeper.short -clears l_hard c -adds l_hard h_vmem 5G
4) qsub -jc sleeper.long -soft -l res_x=3
```

The first job that is submitted (1) will be derived from the sleeper.default job class variant but this job will get the name `MySleeper`.

Job (2) uses the sleeper.short job class but the job name is adjusted. The `-clearp` switch will remove the job name that is specified in the job class. Instead it will get the default job name that would have been assigned without specifying the name in any explicit way. This will be derived from the last part of the script command that will be executed. This script is `/bin/sleep`. So the job name of the new job will be `sleep`.

When job (3) is created the list of hard resource requirements is adjusted. The resource request c is removed and the `h_vmem=5G` resource request is added.

During the submission of job (4) The list of soft resource request is completely redefined. The use of the `-l` will completely replace already defined soft resource requests if any have been defined.

Please note that it is not allowed to trigger operations that would violate any access specifiers. In consequence, the following commands would be rejected:

```
5) qsub -jc sleeper -hard -l res_x 3 (This would remove the a and b resource requests)
6) qsub -jc sleeper /bin/my_sleeper 61 (Neither CMDNAME nor the CMDARGs are modifiable)
```

5.7 Status of Job Classes and Corresponding Jobs

The `-fjc` switch of the `qstat` command can be used to display all existing job classes and jobs that have been derived from them.

```
> qstat -fjc
job class                                                          O  U  states
--------------------------------------------------------------------------------
sleeper.default                                                       X
```

```
  42145 0.55500 Sleeper    user         r     05/15/2012 15:30:47  1
  42146 0.55500 Sleeper    user         r     05/15/2012 15:30:47  1
  42147 0.55500 Sleeper    user         r     05/15/2012 15:30:47  1
  42148 0.55500 Sleeper    user         r     05/15/2012 15:30:47  1
--------------------------------------------------------------------------------
sleeper.long                                                       X d
--------------------------------------------------------------------------------
sleeper.short                                                      X
  42149 0.55500 ShortSleep user        r     05/15/2012 15:30:57   1
  42150 0.55500 ShortSleep user        r     05/15/2012 15:30:57   1
  42151 0.55500 ShortSleep user        r     05/15/2012 15:30:57   1
--------------------------------------------------------------------------------
template.default
```

The O column shows if the user executing the qstat command is the owner of the job class and the U-column is tagged with an X if the corresponding job class can be used by that user to derive new jobs.

The states column will show the character d if the corresponding job class variant is in disabled state and a c if the class is in the configuration conflict state. In all other cases the column will be empty. This indicates that the job class variant can be used to create a new job.

6 Monitoring and Controlling Jobs

6.1 Getting Status Information on Jobs

The command line tool **qstat** delivers all the available status information for jobs. `qstat` supplies various possibilities to present the available information.

Table 14: The Most Common Ways to Use qstat

Command	Description
`qstat`	Without options, `qstat` lists all jobs but without any queue status information.
`qstat -f`	The `-f` option causes qstat to display a summary information of all cause including its load accompanied by the list of all queued as also all pending jobs.
`qstat -ext`	The `-ext` option causes qstat to displays usage information and the ticket consumption of each job.
`qstat -j <job_id>`	The `-j` option causes qstat to display detailed information of a currently queued job.

Examples:

```
# qstat
```

```
job-ID prior name   user   state   submit/start at     queue   slots ja-task-ID
-------------------------------------------------------------------------------
  4 0.55500  job1   user1    r     04/28/2011 09:35:34 all.q@host1      1
  5 0.55500  job2   user1    r     04/28/2011 09:35:34 all.q@host2      1
  6 0.55500  job3   user1    r     04/28/2011 09:35:34 all.q@host2      1
```

```
# qstat -f
queuename                     qtype resv/used/tot. load_avg arch      states
---------------------------------------------------------------------------------
all.q@host1                   BIPC  0/3/10         0.04     lx-amd64
 16 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
 18 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
 23 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
---------------------------------------------------------------------------------
all.q@host2                   BIPC  0/3/10         0.04     lx-x86
 15 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
 19 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
 22 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
---------------------------------------------------------------------------------
all.q@host3                   BIPC  0/3/10         0.04     sol-amd64
 14 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
 17 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
 21 0.55500 Sleeper    user1        t     04/28/2011 09:36:44     1
---------------------------------------------------------------------------------
all.q@host4                   BIPC  0/3/10         1.35     lx-amd64
 20 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
 24 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
 25 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
```

It is also possible to be informed by the Altair Grid Engine system via mail on the status change of a job. To use this feature it necessary to set the `-m` option while submitting the job. This option is available for `qsub`, `qsh`, `qrsh`, `qlogin` and `qalter`.

Table 15: Mail Options to Monitor Jobs

Option	Description
b	Send mail at the beginning of a job.
e	Send mail at the end of a job.
a	Send mail when job is aborted or rescheduled.
s	Send mail when job is suspended.
n	Send no mail (default).

Example: Altair Grid Engine will send mail at the beginning as well as the end of the job:

```
# qsub -m be test_job.sh
```

6.2 Deleting a Job

To delete a job, the **qdel** binary is used.

Table 16: Optional qdel Parameters

Parameter	Description
-f <job_id[s]>	Forces the deletion a job even if the responsible execution host does not respond.
<job_id> -t <range>	Deletes specific tasks of an array job. It is also possible to delete a specific range of array jobs.
-u <user_list>	Deletes all job of the specified user.

The behavior of how Altair Grid Engine handles a forced deletion can be altered by using the following qmaster parameters. This option can be set via **qconf -mconf** as **qmaster_params**.

Table 17: qmaster Parameters for Forced Job Deletion

Parameter	Description
ENABLE_FORCED_QDEL	If this parameter is set, users are allowed to force job deletion on their own jobs. Otherwise only the Altair Grid Engine managers are allowed to perform those actions.
ENABLE_FORCED_QDEL_IF_UNKNOWN	If this parameter is set, qdel <job_id> will automatically invoke a forced job deletion if the host, where the job is running, is of unknown status.

Examples:

Delete all jobs in the cluster (only possible for Altair Grid Engine managers):

```
# qdel -u "*"
```

Delete tasks 2-10 out of array job with the id 5:

```
# qdel 5 -t 2-10
```

Forced deletion of jobs 2 and 5:

```
# qdel -f 2 5
```

6.3 Re-queuing a Job

A job can be rescheduled only if its `rerun` flag is set. This can be done either at time of submission via the `-r` option of qsub, or belatedly via the `-r` option of **qalter** as well as via the `rerun` configuration parameter for queues. This `rerun` configuration can be set with **qconf -mq <queue_name>**.

Examples:

```
# qsub -r yes <job_script>
# qalter -r yes <job_id>
```

There are two different ways to reschedule jobs.

Examples:

Reschedule a job:

```
# qmod -rj <job_id[s]>
```

Reschedule all jobs in a queue:

```
# qmod -rq <queue|queue_instance>
```

Rescheduled jobs are designated Rr (e.g. shown by qstat).

Example:

```
# qstat -f
queuename                      qtype resv/used/tot. load_avg arch       states
---------------------------------------------------------------------------------
all.q@host1                    BIPC  0/2/10         0.01     lx-amd64
    53 0.55500 Sleeper     user1        Rr    05/02/2011 15:31:10     2
---------------------------------------------------------------------------------
all.q@host2                    BIPC  0/2/10         0.01     lx-x86
    53 0.55500 Sleeper     user1        Rr    05/02/2011 15:31:10     2
---------------------------------------------------------------------------------
all.q@host3                    BIPC  0/1/10         0.03     sol-amd64
    53 0.55500 Sleeper     user1        Rr    05/02/2011 15:31:10     1
---------------------------------------------------------------------------------
all.q@host4                    BIPC  0/0/10         0.06     lx-amd64
```

6.4 Modifying a Waiting Job

To change attributes of a pending job **qalter** is used.

qalter is able to change most of the characteristics of a job even those which were set as embedded flags in the script files. Consult the submit(1) main page in regards to the options that can be altered (e.g. the job script).

6.4.1 Altering Job Requirements

It is also possible to alter the requirements of a pending job which have been defined via the -l flag at time or submission.

Example:

Submit a job to host1

```
# qsub -l h=host1 script.sh
```

Alter the host-requirement of this job (with the assumed job-id 45) to host2

```
# qalter -l h=host2 45
```

Note

By altering requested requirements the with -l, keep in mind that the requirements become the new requirements thus the requirements which do not require change must be re-requested.

Example:

Submit a job with the requirement to run on host1 and and on queue2:

```
# qsub -l h=host1,q=queue2 script.sh
```

Alter the host-requirement of this job (with the assumed job-id 45) to host5 and re-request queue2 as requirement

```
# qalter -l h=host5,q=queue2 45
```

If queue2 is NOT stated in the qalter-call, the job will run on any available queue in host5.

6.5 Changing Job Priority

To change the priority of a job the -p option of qalter can be used. It is possible to alter the priority within the range between -1023 and 1024 whereas a negative number decreases priority and a positive one to increases it. If not submitted differently, the default priority is 0. As previously mentioned, a user can only alter his own jobs and in this case, a user is only able to decrease the priority of a job. To increase the priority, the user needs to be either Altair Grid Engine administrator or Altair Grid Engine manager.

Examples:

Increase the job priority of job 45:

```
# qalter -p 5 45
```

Decrease the job priority of 45:

```
# qalter -p -5 45
```

6.6 Obtaining the Job History

To get the history of a job and its accounting information use **qacct**.

qacct parses the accounting file written by *qmaster* and lists all available information for a given job. This includes accounting data such as wall-clock time, cpu-time or memory consumption as also the host where job ran and e.g. the exit-status of the job script. The default Altair Grid Engine accounting file resists in <sge_root>/<cell>/common/accounting. See **accounting(5)** for more information e.g. how the file is composed and what information is stored in it.

Example: Show the accounting information of job 65:

```
# qacct -j 65
==============================================================
qname        all.q
hostname     host1
group        users
owner        user1
project      NONE
department   defaultdepartment
jobname      Sleeper
jobnumber    65
taskid       undefined
account      sge
priority     0
qsub_time    Mon May  9 14:27:32 2011
start_time   Mon May  9 14:27:35 2011
end_time     Mon May  9 14:28:20 2011
granted_pe   mytestpe
slots        5
failed       0
exit_status  0
ru_wallclock 45
ru_utime     0.026
ru_stime     0.019
ru_maxrss    1856
ru_ixrss     0
ru_ismrss    0
ru_idrss     0
ru_isrss     0
ru_minflt    10649
ru_majflt    0
ru_nswap     0
ru_inblock   0
ru_oublock   24
ru_msgsnd    0
ru_msgrcv    0
ru_nsignals  0
ru_nvcsw     101
```

```
ru_nivcsw    26
cpu          0.045
mem          0.000
io           0.000
iow          0.000
maxvmem      17.949M
arid         undefined
```

7 Other Job Types

7.1 Array Jobs

Array jobs are, as mentioned in Types of Workloads being Managed by Altair Grid Engine, those that start a batch job or a parallel job multiple times. Those simultaneously-run jobs are called tasks. Each job receives an unique ID necessary to identify each of them and distribute the workload over the array job.

Submit an array job:

The default output- and error-files are `job_name.[o|e]job_id` and `job_name.[o|e]job_id.task_id`. This means that Altair Grid Engine creates an output- and an error-file for each task plus one for the super-ordinate array-job. To alter this behavior use the -o and -e option of qsub. If the redirection options of qsub are use (-o and/or -e), the results of the individual will be merged into the defined one.

Table 18: Available Pseudo Environment Variables

Pseudo env variable	Description
$USER	User name of the submitting user
$HOME	Home directory of the submitting user
$JOB_ID	ID of the job
$JOB_NAME	Name of the job
$HOSTNAME	Hostname of the execution host
$SGE_TASK_ID	ID of the array task

The -t option of qsub indicates the job as an array job. The -t option has the following syntax:

```
qsub -t n[-m[:s]] <batch_script>
```

-t Option Syntax

- n - indicates the start-id.
- m - indicates the max-id.
- s - indicates the step size.

Examples:

qsub -t 10 array.sh - submits a job with 1 task where the task-id is 10.

qsub -t 1-10 array.sh - submits a job with 10 tasks numbered consecutively from 1 to 10.

qsub -t 2-10:2 array.sh - submits a jobs with 5 tasks numbered consecutively with step size 2 (task-ids 2,4,6,8,10).

Besides the pseudo environment variables already mentioned, the following variables are also exposed which can be used in the script file:

Table 19: Pseudo Environment Variables Available for Scripts

Pseudo env variable	Description
$SGE_TASK_ID	ID of the array task
$SGE_TASK_FIRST	ID of the first array task
$SGE_TASK_LAST	ID of the last array task
$SGE_TASK_STEPSIZE	step size

Example of an array job script:

```
#!/bin/sh

# redirect the output-file of the batch job
#$ -o /tmp/array_out.$JOB_ID
# redirect the error-file of the batch job
#$ -e /tmp/array_err.$JOB_ID

# starts data_handler with data.* as input file
/tmp/data_handler -i /tmp/data.$SGE_TASK_ID
```

Alter an array job:

It is possible to change the attributes of array jobs. But the changes will only affect the pending tasks of an array job. Already running tasks are untouched.

Array job concurrency

The maximum number of concurrently running tasks of an array job can be limited via the -tc switch of qsub (see -tc in submit(1)).

So called concurrent array jobs are jobs where either all tasks can be started in one scheduling interval or no task is started at all (the whole job stays pending). A concurrent array job is submitted using the -tcon switch of qsub (see -tcon in submit(1)). Immediate concurrent array jobs (qsub -tcon y -now y) will be rejected if not all tasks can be started immediately.

Configuration variables (see sge_conf(5)):

- *max_aj_instances* indicates the maximum number of instances of an array job which can run simultaneously.

- *max_aj_tasks* indicates the maximum number of tasks a array job can have.
- *qmaster_params MIN_PENDING_ENROLLED_TASKS* can be used to define for how many pending array tasks individual per task tickets are calculated per job by the Altair Grid Engine policy engine.
- *qmaster_params MAX_TCON_TASKS* is used to limit the number of tasks a concurrent array job can have, value 0 (default) disables concurrent array jobs.

Example:

Submit a job with 20 tasks but only 10 of then can run concurrently. qsub -t 1-20 -tc 10 array.sh

7.2 Interactive Jobs

Usually, Altair Grid Engine uses its own built-in mechanism to establish a connection to the execution host. It is possible to change this to e.g. ssh or telnet, of course.

Configuration variable	Description
qlogin_command	Command to execute on local host if qlogin is started.
qlogin_daemon	Daemon to start on execution host if qlogin is started.
rlogin_command	Command to execute on local host if qrsh is started **without** a command name as argument to execute remotely.
rlogin_daemon	Daemon to start on execution host if qrsh is started **without** a command name as argument to execute remotely.
rsh_command	Command to execute on local host if qrsh is started **with** a command name as argument to execute remotely.
rsh_daemon	Daemon to start on execution host if qrsh is started **with** a command name as argument to execute remotely.

Example of a qlogin configuration:

```
qlogin_command   /usr/bin/telnet
qlogin_daemon    /usr/sbin/in.telnetd
```

The configured commands (qlogin_command, rlogin_command and rsh_command) are started with the execution host, the port number and, in case of rsh_command, also the command name to execute as arguments.

Example:

```
/usr/bin/telnet exec_host 1234
```

Consult sge_conf(5) for more information.

Note

Interactive jobs are not support from or to Windows hosts. One exception is the `qrsh with command`, e.g. `qrsh hostname`. This works also from and to Windows hosts.

7.2.1 qrsh and qlogin

qrsh without a command name as argument and qlogin submit an interactive job to the queuing system which starts a remote session on the execution host where the current local terminal is used for I/O. This is similar to rlogin or a ssh session without a command name.

qrsh with a command executes the command on the execution host and redirects the I/O to the current local terminal. By default, qrsh with command does not open a pseudo terminal (PTY), other than qlogin and qrsh without command, on the execution host. It simply pipes the in- and output to the local terminal. This behavior can by changed via the *-pty* yes option as there are applications that rely on a PTY.

Those jobs can only run in INTERACTIVE queues unless the jobs are not explicitly marked as non-immediate job using the *-now* no option.

7.2.2 qmake

qmake facilitates the possibility to distribute Makefile processing in parallel over the Altair Grid Engine. It is based on GNU Make 3.78.1. All valid options for qsub and qrsh are also available for qmake. Options which has to be passed to GNU Make has to be placed after the "–"-separator.

Syntax:

```
qmake [ options ] -- [ gmake options ]
```

Typical examples how to use qmake:

```
qmake -cwd -v PATH -pe compiling 1-10 -- -debug
```

This call changes the remote execution host into the current working directory, exports the **$PATH** environment variable and requests between 1 and 10 slots in the parallel environment *compiling*. This call is listed as one job in the Altair Grid Engine system.

This means that Altair Grid Engine starts up to 10 qrsh sessions depending on available slots and what is needed by GNU Make. The option -debug will, as it is after the "–"-separator, be passed to the GNU Make instances.

As there is no special architecture requested, Altair Grid Engine assumes the one set in the environment variable **$SGE_ARCH**. If it is not set, qmake will produce a warning and start the make process on any available architecture.

```
qmake -l arch=lx26-amd64 -cwd -v PATH --
```

Other than the example above, qmake is not bound to a parallel environment in this case. qmake will start an own qrsh job for every GNU Make rule listed in the Makefiles.

Furthermore, qmake support two different modes of invocation:

- Interactive mode: qmake invoked by command line implicitly submits a qrsh-job. On this master machine the parallel make procedures will be started and qmake will distribute the make targets and steps to the other hosts which are chosen.
- Batch mode: If qmake with the **–inherit** option is embedded in a simple batch script the qmake process will inherit all resource requirements from the calling batch job. Eventually declared parallel environments (pe) or the `-j` option in the qmake line within the script will be ignored.

Example:

```
#!/bin/sh
qmake --inherit --
```

Submit:

```
qsub -cwd -v PATH -pe compiling 1-10 <shell_script>
```

7.2.3 qsh

qsh opens a **xterm** via an interactive X-windows session on the execution host. The display is directed either to the X-server indicated by the *$DISPLAY* environment variable or the one which was set by the *-display* qsh command line option. If no display is set, Altair Grid Engine tries to direct the display to 0.0 of the submit host.

7.3 Parallel Jobs

A parallel job runs simultaneously across multiple execution hosts. To run parallel jobs within the Altair Grid Engine system it is necessary to set up **parallel environments** (pe). It is customary is to have several of such parallel environments e.g. for the different MPI implementations which are used or different ones for tight and loose integration. To take advantage of parallel execution, the application has to support this. There are a dozen software implementations that support parallel tasks like *OpenMPI, LAM-MPI, MPICH* or *PVM*. supports two different ways of executing parallel jobs:

- Loose Integration

Altair Grid Engine generates a custom machine file listing all execution hosts chosen for the job. Altair Grid Engine does not control the parallel job itself and its distributed tasks. This means that there is no tracking of resource consumption of the tasks and no way to delete runaway tasks. However, it is easy to set up and nearly all parallel application technologies are supported.

- Tight Integration

Altair Grid Engine takes control of the whole parallel job execution. This includes spawning and controlling of all parallel tasks. Unlike the Loose Integration Altair Grid Engine is able to track the resource usage correctly including all parallel tasks as also to delete runaway tasks via qdel. However the parallel applications has to support the tight Altair Grid Engine integration (e.g. OpenMPI which has to be built with *–enable-sge*).

7.3.1 Parallel Environments

Setup a parallel environment

```
qconf -ap my_parallel_env
```

This will create a parallel environment with the name `my_parallel_env`. In the opening editor it is possible to change the properties of the pe.

Property Description

pe_name The name of the parallel environment. This one has to be specified at job submission.

slots The maximum number of slots which can be used/requested concurrently.

user_lists User-sets which are allowed to use this pe. If NONE is set, everybody is allowed to use this pe.

xuser_lists User-sets which are **not** allowed to use this pe. If NONE is set, everybody is allowed to use this pe.

start_proc_args This command is started prior the execution of the parallel job script.

stop_proc_args This command proceeds the execution of the parallel job script finished.

per_pe_task_prolog This command is started prior the execution of any parallel slave task.

per_pe_task_epilog This command is started after a parallel slave task finished.

allocation_rule The allocation rule is interpreted by the scheduler and helps to determine the distribution of parallel processes among the available execution hosts. There are three different rules available:
- **<int>**: This defines the number of max processes allocated at each host.
- **$fill_up**: All available slots on a host will be used (filled up). If there are no more slots available on this particular host, the remaining processes will be distributed to the next host.
- **$round_robin**: All processes of a parallel job will be uniformly distributed of the Altair Grid Engine system.

control_slaves This options is in control when the parallel environment is loose or tightly integrated.

job_is_first_task This parameter indicates if the job submitted already contains one of the parallel tasks.
With the introduction of per task requests (`-petask` submit option) it is advised to not set this property to `FALSE` anymore.

urgency_slots For pending jobs with a slot range pe request, the number of slots is not determined. This setting specifies the method to be used by Altair Grid Engine to assess the number of slots such jobs might finally get. These methods are available:
- **<int>**: This integer number is used as prospective number of slots.
- **min**: The slot range minimum is used as prospective number of slots.
- **max**: The slot range maximum is used as prospective number of slots.
- **avg**: The average of all numbers occurring within the job's pe range request is assumed.

accounting_summary If set to TRUE, the accounting summary of all tasks are combined in one single accounting record otherwise every task is stored in an own accounting record. This option is only considered if control_slaves is also set.

daemon_forks_slaves If this parameter is TRUE, a single daemon is started via `qrsh -inherit` on every slave host which forks the slave tasks (value TRUE, e.g. used for openmpi or lam integration). To use this parameter **control_slaves** has to be TRUE.

master_forks_slaves This parameter can be set to TRUE if the master task (e.g. mpirun called in the job script) starts tasks running on the master host via fork/exec instead of starting them via `qrsh -inherit`. To use this parameter **control_slaves** has to be TRUE. ————————————————————————————————————-

: Properties of the Parallel Environment (PE)

These properties will additionally be shown when using `qconf -sp`:

Table 22: Read-only properties of the Parallel Environment (PE)

Property	Description
used_slots	The number of currently occupied slots of this parallel environment. This is a read-only value and will only show with `qconf -sp`!
bound_slots	The number of currently occupied slots of this parallel environment that got preempted, but are not yet available. This is a read-only value and will only show with `qconf -sp`!

Table 23: Examples and Templates for MPI and PVM

Example/Template	Parallel Environment
/SGE_ROOT/mpi/	MPI and MPICH
/SGE_ROOT/pvm/	PVM

See sge_pe(5) for detailed information.

7.3.2 Submitting Parallel Jobs

Parameter Description

-pe parallel_environment This parameter indicates that this is a parallel job. **n[-[m]][-]m**

Note
For declaring the parallel_environment the wildcard character * is allowed (e.g. mpi*).

```
**Allowed Range Specifications for Job**

**n-m**
Minimum n slots and Maximum m slots

Example: 2-10

**m**

This is an abbreviation for m-m. Exactly m slots
 are needed.

Example: 10

**-m**

This is an abbreviation for 1-m.

Example: -10

**n-**

At least n slots are needed but as much as possible
slots are wanted.

Example: 10-
```

-petask tid_range_list . . . Allows to define for a range of parallel tasks specific resource and queue requests. **tid_range_list** "tid_range_list" := "tid_range" [, "tid_range", . . .] | 'master' | 'slaves' "tid_range" := n ['-' [m] [':'s]] where n and m are the lower and upper PE task ID and s is the step size.

```
Examples for tid_range_lists:<br>
-petask 0
-petask master
```

```
Both denote the master task.

-petask 1-
-petask slaves
Both denote all slave tasks.

-petask 1-:2
Every second PE task starting with PE task ID 1, i.e. 1,3,5,7,...

-petask 1-7:2,6
The PE tasks IDs 1,3,,5,6,7
```

-masterq queue Allows to define on which queue the master task has run. This parameter is deprecated, use **-petask master -q queue** instead.

-masterl requests Allows to define resource requests for the master task. This parameteris deprecated, use **-petask master -l requests** instead.

: Parameters to Submit a Parallel Job
Example:
qsub -pe mpi_pe 4-10 -petask master -q super.q mpi.sh
See submit(1) for more information.
Parallel Jobs and Core Binding

Note
Core Binding is not supported on Windows execution hosts.

Note
The behavior of the core-binding command for PE-jobs changed with Altair Grid Engine version 8.6! The amount of cores to bind specified during submission changed from meaning "per host" to mean "per PE-task". For more information, see the section [PE-Jobs with core binding][PE-Jobs with core binding]

Parallel jobs can exploit the core binding feature in different ways. The following sections provides an overview of there different methods which can be used.
Using the -binding pe Request
One possibility of assigning CPU cores to a job is using the "pe" flag of the binding option itself. The following example demonstrates requesting two cores per host, as well as two slots per host, on all two hosts where the parallel job runs.
qsub -binding pe linear:2 -pe fixed2 4 ...
Note that the parallel environment fixed2 contains following fixed allocation rule:
allocation_rule 2

The allocation rule enforces the scheduler to select two slots per host, while the binding request enforces the scheduler to select 2 free cores per host.
After dispatching the parallel job, the selected cores are marked as used in the scheduler. This can be displayed using the qhost -F m_topology_inuse topology string.
The selected cores of a specific parallel job are displayed in the **qstat -j <jobno>** output in the binding output.
binding 1: host_10=0,0:0,1, host_12=0,0:0,1
This means that on host_10 the job got core 0 and core 1 on the socket 0 and on host_12 the same core selection was done.
With using the **-binding pe** option the scheduler does its decision and marks those cores as used but on the execution side no real core binding done (in contrast to the **-binding set** (which equals just **-binding**) option. What Altair Grid Engine does is that it writes its decision to the **pe_hostfile** in the last column. This file is usually exploited by tight parallel jobs integration.
In the example it looks as follows:
host_10 2 all.q@macsuse 0,0:0,1 host_12 2 all.q@u1010 0,0:0,1
Using these <socket,core> pairs which are separated by a ":" sign the parallel job can exploit the information and bind the parallel jobs on these cores. Note, that when having multiple queue instances on a host and the parallel job spans over different queue instances on the same host, that multiple entries for one host in the "pe_hostfile" exists. Since the binding is a "per host" decision (as it is a per host request) all decisions for on particular host but different queue instances on that host are the same. Since version 8.1. different decisions for different hosts can be made. Hence a "pe_hostfile" can also look like below.
host_10 2 all.q@macsuse 1,2:1,2 host_12 2 all.q@u1010 0,0:0,1
One example how to exploit this information to bind the parallel tasks on different cores is using the "rankfile" of OpenMPI. With the rankfile it can be controlled how OpenMPI binds each individual rank to a separate core. This can massively improve the performance of OpenMPI. Like for other tight integrations such a rankfile must be created based on the "pe_hostfile" information. Altair Grid Engine contains an example in the `$SGE_ROOT/mpi/openmpi_rankfile` product directory.
Using the SGE_BINDING Environment Variable
Jobs with Core Binding

Note

The output of qstat -j changed in 8.1 with respect to the final binding done per job task. Before just one topology string was reported (for the master task), since 8.1 core bindings on all hosts where the parallel job runs are showed as lists of , tuples.

Note
Since version 8.1 regardless of the binding mode (env, pe, or set) the SGE_BINDING environment variable will always be available.

Note
Core Binding is not supported on Windows execution hosts.

Today's execution hosts are usually multi-socket and multi-core systems with a hierarchy of different caches and a complicated internal architecture. In many cases it is possible to exploit the execution host's topology in order to increase the user application performance and therefore the overall cluster throughput. Another important use case is to isolate jobs on the execution hosts from another in order to guarantee better run-time stability and more fairness in case of over-allocation of the host with execution threads. The Altair Grid Engine provides a complete subsystem, which not just provides information about the execution host topology, it also allows the user to force the application to run on specific CPU cores. Another use is so that the administrator can ensure via JSV scripts that serial user jobs are using just one core, while parallel jobs with more granted slots can be run on multiple CPU cores. In Altair Grid Engine core binding on Linux execution hosts is turned on by default, while on Solaris hosts it must be enabled per execution host by the administrator (see Enabling and Disabling Core Binding).

Note
Run the utilbin/<ARCH>/loadcheck -cb command in order to figure out the support of core binding on the specific execution hosts.

In Altair Grid Engine version 8.1 the component, which is responsible for core selection on execution hosts was moved from the execution host component into the scheduler component. Hence it is possible now to guarantee a specific binding for a job because the scheduler searches just for hosts which can fulfill the requested binding.
Showing Execution Host Topology Related Information
By default, the qhost output shows the number of sockets, cores and hardware supported threads on Linux kernel versions 2.6.16 and higher and on Solaris execution hosts:
> qhost HOSTNAME ARCH NCPU NSOC NCOR NTHR LOAD MEMTOT MEMUSE SWAPTO SWAPUS ———————————————————————————————— global - - - - - - - - - host1 lx-amd64 1 1 1 1 0.16 934.9M 150.5M 1004.0M 0.0 host2 lx-amd64 4 1 4 1 0.18 2.0G 390.8M 2.0G 0.0 host3 lx-amd64 1 1 1 1 0.06 492.7M 70.2M 398.0M 0.0

There are also several topology related host complexes defined after an Altair Grid Engine standard installation:
> qconf -sc … m_core core INT <= YES NO 0 0 NO 0.000000 YES YES m_socket socket INT <= YES NO 0 0 NO 0.000000 YES YES m_thread thread INT <= YES NO 0 0 NO 0.000000 YES YES m_topology topo RESTRING == YES NO NONE 0 NO 0.000000 YES YES m_topology_inuse utopo RESTRING == YES NO NONE 0 NO 0.000000 YES YES m_topology_numa numa RESTRING == YES NO NONE 0 NO 0.000000 YES YES m_cache_l1 mcache1 MEMORY <= YES NO 0 0 NO 0.000000 YES YES m_cache_l2 mcache2 MEMORY <= YES NO 0 0 NO 0.000000 YES YES m_cache_l3 mcache3 MEMORY <= YES NO 0 0 NO 0.000000 YES YES m_numa_nodes nodes INT <= YES NO 0 0 NO 0.000000 YES YES
The host specific values of the complexes can be shown in the following way:
> qstat -F m_topology,m_topology_inuse,m_socket,m_core,m_thread queuename qtype resv/used/tot. load_avg arch states
——————————————————————————————— all.q@host1 BIPC 0/0/10 0.00 lx26-amd64 hl:m_topology=SC hl:m_topology_inuse=SC hl:m_socket=1 hl:m_core=1 hl:m_thread=1 ——————————————————————————————— all.q@host2 BIPC 0/0/10 0.00 lx26-amd64 hl:m_topology=SCCCC hl:m_topology_inuse=SCCCC hl:m_socket=1 hl:m_core=4 hl:m_thread=4
——————————————————————————————— all.q@host3 BIPC 0/0/10 0.00 lx26-amd64 hl:m_topology=SC hl:m_topology_inuse=SC hl:m_socket=1 hl:m_core=1 hl:m_thread=1
`m_topology` and `m_topology_inuse` are topology strings. They encode sockets `(S)`, cores `(C)`, and hardware supported threads `(T)`. Hence `SCCCC` denotes one socket host with a quad core CPU and `SCTTCTTSCTTCTT` would encode a two socket system with a dual-core CPU on each socket, which supports hyperthreading. The difference between the two strings is that `m_topology` remains unchanged, even when core bound jobs are running on the host, while `m_topology_inuse` displays the cores, which are currently occupied (with lowercase letters). For example `SccCC` denotes a quad-core CPU, which has two jobs bound on the first and second core, if each job requested only one core.
`m_socket` denotes the number of sockets on the host. `m_core` is the total number of cores, the host offers. `m_thread` is the total number of hardware supported threads the host offers. `m_topology_numa` is an enhanced topology string. In addition to the `S`, `C`, and `T` keywords there are `[` and `]` characters which are marking a specific NUMA node on the execution host. A NUMA (non-uniform memory access) node is a particular area for which the memory latency is the same (usually it is per socket memory).
Requesting Execution Hosts Based on the Architecture
In order to request specific hosts for a job, all the complexes described in the sub-section above can be used. Because and are regular expression strings (type RESTRING) special symbols like * can be used as well. In the following example a quad core CPU is requested:
> qsub -b y -l m_topology=SCCCC sleep 60
This does not correspond to:
> qsub -b y -l m_core=4,m_socket=1 sleep 60
Because the latter request does also match to a hexacore or higher CPU because m_core is defined as “<=”. In order to get a host with a free (currently unbound) quadcore CPU:
> qsub -b y -l m_topology_inuse=SCCCC sleep 60

In order to get a host with at least one quad core CPU, which is currently not used by a core bound job:
> qsub -b y -l m_topology_inuse="*SCCCC*" sleep 60
Requesting Specific Cores

Note
Topology selections (socket/core selections) are not part of a resource reservation yet. Hence jobs submitted with a specific binding and -R y might not be started even when a reservation was done. This can be prevented when using the -binding linear request and aligning the amount of slots per host to the amount of cores per host.

Altair Grid Engine supports multiple schemata in order to request cores on which the job should be bound. Several adjoined cores can be specified with the `linear:<amount>` request. In some cases it could be useful to distribute the job over sockets, this can be achieved with the `striding:<stepsize>:<amount>` request. Here the `stepsize` denotes the distance between two successive cores. The stepsize can be aligned with a `m_topology request` in order to get the specific architecture. The most flexible request schema is `explicit:<socket,core>[:<socket,core>[...]]`. Here the cores can be selected manually based on the socket number and core number.
Examples:
Bind a job on two successive cores if possible:
> qsub -b y -binding linear:2 sleep 60
Request a two-socket dual-core host and bind the job on two cores, which are on different sockets:
> qsub -b y -l m_topology=SCCSCC -binding striding:2:2 sleep 60
Request a quad socket hexacore execution host and bind the job on the first core on each socket:
> qsub -b y -l m_topology=SCCCCCCSCCCCCCSCCCCCCSCCCCCC -binding explicit:0,0:1,0:2,0:3,0 sleep 60
PE-Jobs with core binding

Note
The behavior of the core-binding command for PE-jobs changed with Altair Grid Engine version 8.6! The amount of cores to bind specified during submission changed from meaning "per host" to mean "per PE-task".

For PE-jobs it is hard or even impossible to know in advance how many tasks are going to be scheduled on how many hosts. Therefore, with Altair Grid Engine version 8.6, the core-binding request changed its behavior to accommodate this fact. The binding-request changed its meaning from being a "per host" request, to being a "per PE-task" request. This means that the requested amount of cores means for a PE-job "per PE-task".

For example, if a job with
> qsub -pe mype 5-7 -binding linear:2 -b y sleep 60
is submitted, it means that each PE-task will get 2 cores, no matter on which host or on how many hosts the tasks are scheduled.
There are different binding-strategies, most of them exist in two versions: "host aware" and "host unaware" strategies. For example, there are two versions of linear binding strategies: `linear` and `linear_per_task`. Host unaware strategies have the suffix "_per_task".
With "host aware" strategies, all tasks that end up on a host have to adhere to the given strategy together. For "host unaware" strategies, each task has to adhere to the strategy on its own. This is less strict and usually more tasks can fit on a host. For example, if a job with
> qsub -pe mype 2-4 -binding striding:2:2 -b y sleep 60
is submitted to a single host with the topology `SCCCCCCCC`, the resulting topology would be `ScCcCcCcC`, with 2 tasks on that host. Here task 1 would get the binding `ScCcCCCCC` and task 2 `SCCCCcCcC`. Both tasks together have to adhere to the strategy `striding` with a step size of 2.
On the other hand, if the same job is submitted with the strategy `striding_per_task`,
> qsub -pe mype 2-4 -binding striding_per_task:2:2 -b y sleep 60
one would get 4 scheduled tasks and the topology `Scccccccc`, where task 1 would get `ScCcCCCCC`, task 2 `SCCCCcCcC`, task 3 `SCcCcCCCC` and task 4 `SCCCCCcCc`. Each task has to adhere to the strategy `striding` on its own, but now they can be interleaved.
With "host aware" strategies, PE-tasks "see each other", when it comes to selecting cores, whilst for "host unaware" strategies, PE-tasks do not "see each other".
For more information, especially on the possible binding strategies and their behavior, see man page submit(1).
NUMA Aware Jobs: Jobs with Memory Binding and Enhanced Memory Management

Note

Only jobs running on lx-amd64 execution hosts are able to be set to use a specific memory allocation strategy. The loadcheck -cb utility will show more information about the capabilities of the execution host. May not work with older Linux kernels or with missing libnuma system library.

Since today's execution hosts are not only multi-core hosts but also having a NUMA architecture there is a need to align jobs with the particular memory allocation strategy. Altair Grid Engine 8.7 allows you to do so by using the **-mbind** submission parameter alone or in combination with the **-binding** parameter as well as with the following memory related complex **m_mem_free**. Advantages can be more stable and under certain circumstances result in faster job run-times and better job isolation. With Altair Grid Engine 8.7 following complexes are additionally created during installation time:

Complex Name	Description
m_topology_numa	The NUMA topology string which displays the NUMA nodes of the specific architecture.

m_mem_free Displays the amount of free memory available on the execution host. Used for requesting NUMA memory globally on host as well as implicitly on the different NUMA nodes, depending on the schedulers decision (source /proc/meminfo and scheduler internal accounting). From the /proc/meminfo file the sum of FreeMem, Buffers, Cached, and SwapCached is taken into account as free memory (since 8.1.4).

m_mem_used Displays the amount of used memory in the host.

m_mem_total Displays the amount of total memory on the host (source /proc/meminfo).

m_mem_free_n0 - Displays the amount of free memory the node (source m_mem_free_n3 /sys/devices/system/node/node0/meminfo and scheduler internal accounting). From the /node/meminfo file the sum of Inactive and Free memory is taken into account as free memory (since 8.1.4).

m_mem_used_n0 - Displays the amount of used memory on the node (total - free). m_mem_used_n3

m_mem_total_n0 - Displays the amount of total memory the node (source m_mem_total_n3 /sys/devices/system/node/node0/meminfo).

m_cache_l1 Amount of level 1 cache on the execution host.

m_cache_l2 Amount of level 2 cache on the execution host.

m_cache_l3 Amount of level 3 cache on the execution host.

7.4 m_numa_nodes Amount of NUMA nodes on the execution host.

: NUMA related complexes

The -mbind parameter has following effect:

Table 27: The -mbind submission parameter

Parameter	Description	Dependencies
-mbind cores	The job prefers memory on local NUMA nodes (default), but the job is also allowed to use memory from other NUMA nodes.	required: -binding optional: -l m_mem_free=<mem_per_slot>
-mbind cores:strict	The job is only allowed allocate memory on the local NUMA node.	see -mbind cores
-mbind round_robin	The memory allocated by the job is provided by the OS in an interleaved fashion.	optional: -l m_mem_free=<mem_per_slot>

Parameter	Description	Dependencies
-mbind nlocal	Sets implicitly core binding as well as memory binding strategy chosen by the scheduler.	**required: -l m_mem_free=<mem_per_slot>** not allowed: -binding

There is a special memory consumable which can be used in conjunction with the **-mbind** parameter: **m_mem_free**. This complex holds the total amount of free memory on all NUMA nodes of the execution host. The value is derived from the actual load reported by the execution host as well as from the load calculated by the scheduler based on the memory requests. The minimum of both values is the observed value of **m_mem_free**. In case the execution host has different NUMA nodes, the memory status of those is shown in the **m_mem_fee_n** complex values. Accordingly, there are complexes showing the total amount of memory per node as well as the used memory per node. After installation the **m_mem_free** consumables are initialized on host level through setting the host `complex_values` field to the specific values. They can be showed by the `qconf -se <exechostname>` command.

Note
Resource reservation with core binding or memory affinity when m_mem_free is used is currently not fully supported. This means that for specific implicit memory requests (memory per NUMA node/socket) no reservation is done.

If the job can't run due to a non-valid binding or missing memory the job can get a reservation on the (not on core or per socket memory resources), but only when the requested memory is lower than the actual amount of memory (m_mem_free). In order to overcome this issue the reporting of m_mem_free as load value can be turned off with execd_params DISABLE_M_MEM_FREE=1 (qconf -mconf).

Depending on the **-mbind** request, the -binding request, the **m_mem_free** request and the amount of **slots** (parallel environment jobs) the scheduler seeks an appropriate execution host, which can fulfill the requests and decrements the amount of memory automatically for the chosen NUMA nodes.

7.4.1 Memory Allocation Strategy round_robin

This memory allocation strategy sets the memory policy of the jobs process into an **interleaved** mode. This means that memory allocations are distributed over different memory regions. If the job is scheduled to hosts which don't support this, the OS default memory allocation is done.

When memory allocation strategy **round_robin** was requested together with the special resource **m_mem_free** then the requested amount of memory is decremented from the **m_mem_free** variable. Additionally the per socket memory (**m_mem_free_n**_N_) is decremented equally from all sockets of the selected execution host.

When it is not possible to distribute the amount of free memory equally (because one or more of the NUMA nodes don't offer that amount of memory), then the host is skipped.

For parallel jobs, when requested **m_mem_free** together with `-mbind round_robin`, the amount of **m_mem_free** actually decremented on a particular host depends on the amount of granted slots on this host and at the same time it limits (the socket with the least amount of free memory) the amount of slots which can be granted, when needed. For example: When 4 slots are granted on a particular host, the amount of **m_mem_free** is multiplied by 4. Hence each socket has to offer (m_mem_free * 4) / <amount_of_NUMA_sockets> bytes free on each socket.

Examples:

```
qsub -mbind round_robin -binding striding:2:4 mem_consuming_job.sh
```

This results in a job which runs on a 2 quad core socket machine with memory affinity set to interleaved (to all memory banks on the host) for best possible memory throughput for certain job types.

```
qsub -mbind round_robin mem_consuming_job.sh
```

This results in a job which runs unbound and takes memory in an interleaved fashion.

```
qsub -mbind round_robin -binding linear:2 -pe mytestpe 4 -l m_mem_free=2G -b y
                                                                  sleep 13
```

Let's assume here that mytestpe has an allocation rule of pe_slots. Then the job is running on a host which offers 4*2GB=8GB of m_mem_free as well as on each NUMA node (m_mem_free_nX) at least 8GB/ free memory. The memory consumable m_mem_free is decrement by 8GB and all consumables representing a NUMA node (m_mem_free_n0 to m_mem_free_nX) are decremented by 8GB/ memory. The same behaviour can be seen when "-binding" strategy is changed to any of the available ones, or even when "-binding" is not selected.

7.4.2 Memory Allocation Strategy cores and cores:strict

This memory allocation strategy takes memory **from all NUMA nodes where the job is bound** to (with core binding) into account. If no core binding (-binding) was requested the job is rejected during submission time. Depending on the parameter the memory request is either restricted to local NUMA nodes (cores:strict) only or **local memory is preferred** *(cores)*.

If the memory request, which comes with the job submission command, can not be fulfilled (because NUMA node N offers not as much memory) the node is skipped by the scheduler. On 64bit Linux internally the system call **mbind** (see man mbind) is executed.

The requested memory (when using -l m_mem_free) is decremented from **m_mem_free** as well as from the NUMA nodes (**m_mem_free_n**_N_) where the job is bound to. When a job gets for example 2 cores on socket 1 and one core on socket 2 then the amount of memory on **m_mem_free_n1** is decremented by the total amount of requested memory divided by the amount of granted cores (here 3) multiplied by the amount of granted cores

on the particular NUMA node (here 2). The consumable **m_mem_free_n2** is charged by half of this amount of memory.

Strict means: Only local memory on NUMA node allowed.

Without any keyword the memory allocation strategy is set on Linux to the "**preferred**" mode, that means the job gets memory from the near node as long as there is free memory. When there is no more free memory it is allowed to use memory from a greater distance.

Examples:

```
qsub -mbind cores -binding linear:1 /bin/sleep 77
```

The job gets bound to a free core. The memory requests are preferred on the same NUMA node. If there is no more memory free the next memory request is taken from a node with an higher distance to the selected core.

```
qsub -mbind cores -binding linear:1 -l m_mem_free=2G /bin/sleep 77
```

The job gets bound to a free core only on a NUMA node which currently offers 2GB. The memory requests are preferred on the same NUMA node. If there is no more memory free the next memory request is taken from a node with an higher distance to the selected core. The requested memory is debited from nX_mem_free consumable (memory job-request / amount of occupied cores on node).

> **Warning**
> This could cause out of memory errors on strict jobs in case of overflows. Hence mixing strict with preferred jobs is not recommended.

```
qsub -mbind cores:strict -binding linear:1 /bin/sleep 77
```

The job gets bound to a free core. The memory is always taken from the local NUMA node. If there is no more memory free on the NUMA node the program gets by the next program break extension (brk()) an out of memory exception.

```
qsub -mbind cores:strict -binding striding:2:4 -pe mytestepe 2 -l m_mem_free=2G
                                                                /bin/sleep 77
```

Complete parallel job requests 2G * 2 slots = 4GB memory and 2 cores on two sockets (quad core processors). Assumption: Each core needs 2 GB. The job gets scheduled to a particular host if both NUMA nodes (here both sockets) offer each 2GB m_mem_free_nX. If not the host is skipped. The particular consumables are decremented by that amount.

```
qsub -mbind cores /bin/sleep 77
```

The job gets rejected because the binding is missing.

7.4.3 Memory Allocation Strategy nlocal

This memory allocation strategy automatically allocates cores and set an appropriate memory allocation strategy for single-threaded or multi-threaded (parallel environments with allocation_rule pe_slots) depending on the memory request and the execution hosts characteristics (free sockets/cores and free memory on the specific NUMA nodes).

Note
Requirements: No core binding request set (otherwise the job is rejected), but a mandatory request for the **m_mem_free consumable**. If this consumable is not requested the job is rejected.

-mbind nlocal with Sequential Jobs

The **nlocal** strategy is intended to use for sequential as well for multi-threaded jobs in order to get stable job run-time results as well highest amount of memory throughput. The only requirement for the jobs is the amount of memory the job needs per slot (-l m_mem_free=). When multiple slots are needed then a parallel environment with the allocation rule "pe_slots" (so that the job is not distributed to different hosts) is required. The behavior is undefined with PEs having other allocation rules configured. The scheduler tries to place jobs on sockets which offers most free cores and have additionally the required amount of memory free on the specific NUMA node (m_mem_free_n<node>). If the required amount of memory is more than each socket has installed the job will run on one socket exclusively if one is completely free (with out any core-bound jobs). If the required memory is more than free memory each NUMA node (socket) can offer, but less than installed memory on the NUMA nodes, the host is skipped. In this scenario the job has either to wait until the required amount of memory is free on this host or it can run an a more appropriate host.

On NUMA execution nodes the scheduler tries to do following for **sequential jobs**:

- If the host can't fulfill the **m_mem_free** request then the host is skipped.
- If the job requests more ram than free on each socket but less than installed on the sockets the host is skipped.
- If memory request is **smaller** than amount of free memory on a socket, try to bind the job to **one core on the socket** and decrement the amount of memory on this socket (m_mem_free_n<nodenumber>). The global host memory m_mem_free on this host is decremented as well.
- If memory request is **greater** than the amount of free memory on any socket, find an unbound socket and bind it there completely and allow memory overflow. Decrement from m_mem_free as well as from m_mem_free_n and the remaining memory round robin from the remaining sockets.
- If both are not possible go to the next host.

-mbind nlocal with Parallel Jobs

Parallel jobs are handled in the scheduler the following way (only pe_slots PEs are supported, the behaviour for other allocation rules is unspecified):

- Hosts that do not offer m_mem_free memory are skipped (of course hosts that do not offer the amount of free slots requested are skipped as well).
- If the amount of requested slots is **greater** than the amount of **cores per socket**. The job is dispatched to the host without any binding.
- If the amount of requested slots is **smaller** than the amount of **cores per socket** do following:
  - If there is any socket which offers enough memory (m_mem_free_n) and enough free cores bind the job to these cores and set memory allocation mode to **cores:strict** (so that only local memory requests can be done by the job).
  - If this is not possible try to find a socket which is completely unbound and has **more** than the required amount of memory **installed** (m_mem_total_n). Bind the job to the complete socket, decrement the memory on that socket at m_mem_free_n (as well as host globally on m_mem_free), and set the memory allocation strategy to **cores** (**preferred** usage of socket local memory).

If nothing matches then the host is skipped.

Other examples

The following example demonstrated how a parallel job with 4 threads (requesting the parallel environment testpe for 4 slots (allocation_rule $pe_slots) each needed 1 gigabyte of memory is submitted (4 gigabytes for the job in total):

```
qsub -mbind cores:strict -binding linear:4 -pe testpe 4 -l m\_mem\_free=1G testjob.sh
```

For this job the scheduler skips all hosts which do not have 4 slots, 4 cores as well as 4 gigabyte free (according to the m_mem_free value). If a host is found it is first tried to accommodate the job on one single socket, if it is not possible then a distribution over the least amount of sockets is tried. If the host does not fulfill the memory request on the chosen socket / NUMA node (m_mem_free_n<node>) the host is discarded. Otherwise the job gets assigned the specific cores as well as the particular amount of memory on the machine as well on the NUMA nodes. Hence a -l m_mem_free request comes with implicit m_mem_free_n requests depending of the binding the scheduler determines.

7.5 Checkpointing Jobs

Note

Checkpointing is not supported on Windows execution hosts.

Checkpointing delivers the possibility to save the complete state of a job and to restart from this point of time if the job was halted or interrupted. Altair Grid Engine supports two kinds of Checkpointing jobs: the user-level and the kernel-level Checkpointing.

7.5.1 User-Level Checkpointing

User-Level Checkpointing jobs have to do their own checkpointing by writing restart files at certain times or algorithmic steps. Applications without an integrated user-level checkpointing can use a checkpointing library like the Condor project.

7.5.2 Kernel-Level Checkpointing

Kernel-Level Checkpointing must be provided by the executing operating systems. The checkpointing job itself does not need to do any checkpointing. This is done by the OS entirely.

7.5.3 Checkpointing Environments

To execute and run checkpointing jobs environments, similar to parallel jobs, are necessary to control how, when and how often checkpointing should be done.

Table 28: Handle Checkpointing Environments with qconf

Parameter	Description
-ackpt	add a checkpointing environment
-dckpt	delete the given checkpointing environment
-mckpt	modify the given checkpointing environment
-sckpt	show the given checkpointing environment

A checkpointing environment is made up of the following parameters:

Parameter Description

ckpt_name
The name of the checkpointing environment. interface
The type of the checkpointing which should be used. Valid types:

```
**hibernator**

The Hibernator kernel-level checkpointing is interfaced.

**cpr**

The SGI kernel-level checkpointing is used.

**cray-ckpt**

The Cray kernel-level checkpointing is used.
```

```
**transparent**

\GEFullName{} assumes that the job submitted within
this environment uses a checkpointing library such as
the mentioned Condor.

**userdefined**

\GEFullName{} assumes that the job submitted within
this environment uses a its private checkpointing method.

**application-level**

Uses all interface commands configured in the
checkpointing object. In case of one of the kernel
level checkpointing interfaces the restart\_command
is not used.
```

ckpt_command
Command which will be executed by Altair Grid Engine to initiate a checkpoint. migr_command
Command which will be executed by Altair Grid Engine during a migration of a checkpointing job from one host to another. restart_command
Command which will be executed by Altair Grid Engine if a previously checkpointed job is restarted. clean_command
Command which will be executed by Altair Grid Engine after a checkpointing job is completed. ckpt_dir
Directory where checkpoints are stored. ckpt_signal
A UNIX signal which is sent by Altair Grid Engine to the job when a checkpoint is initiated. when
Point of time when checkpoints are expected to be generated. Valid values for this parameter are composed by the letters s, m, x and r and any combinations thereof without any separating character in between: **s** The job is checkpointed, aborted and if possible migrated if the corresponding execution daemon is shut down on the job's machine. **m** Checkpoints are generated periodically at the min_cpu_interval interval defined by the queue in which a job executes. **x** A job is checkpointed, aborted and if possible, migrated as soon as the job is suspended (manually as well as automatically). **r** A job is rescheduled (not checkpointed) when the job host goes into an unknown state and the time interval reschedule_unknown defined in the global/local cluster configuration is exceeded. ——- : Handle Checkpointing Environments Parameters

7.5.4 Submitting a Checkpointing Job

```
# qsub -ckpt <ckpt_env> -c <when_options> job
```

The -c option is not mandatory. It can be used to override the when parameters stated in the checkpointing environment.

Example of a Checkpointing Script

The environment variable **RESTARTED** is set for checkpointing jobs that are restarted. This variable can be used to skip e.g. preparation steps.

```
#!/bin/sh
#$ -S /bin/sh

# Check if job was restarted/migrated
    if [ $RESTARTED = 0 ]; then
# Job is started first time. Not restarted.
    prepare\_ckpt\_env
    start\_job
else
# Job was restarted.
    restart\_job
fi
```

7.6 Immediate Jobs

Altair Grid Engine tries to start such jobs immediately or not at all. If, in case of array jobs, not all tasks can be scheduled immediately, none will be started. To indicate an immediate job, the -now option has to be declared with the parameter yes.

Example:

```
# qsub -now yes immediate_job.sh
```

The -now option is available for qsub, qsh, qlogin and qrsh. In case of qsub no is the default value for the -now option, in case of qsh, qlogin and qrsh vice versa.

7.7 Reservations

With the concept of Advance Reservations (AR) it is possible to reserve specific resources for a job, an user or a group in the cluster for future use. If the AR is possible (resources are available) and granted it is assigned an ID.

With Standing Reservations the allocation of recurring Advance Reservations can be scheduled. Standing Reservations are defined through a weekly calendar which determines when Advance Reservations start and when they end. The Advance Reservations within a Standing Reservation behave like normal Advance Reservations with the difference that all Advance Reservations have the same AR ID and that waiting jobs requesting that AR ID are not deleted when one Advance Reservation ends. They are only deleted at the end of the last occurrence of an Advance Reservation.

7.7.1 Advance Reservations

Configuring Advance Reservations To be able to create advance reservations the user has to be member of the arusers list. This list is created during the Altair Grid Engine installation. Use qconf to a user to the arusers list.

```
# qconf -au username arusers
```

Creating Advance Reservations `qrsub` is the command used to create advance reservations and to submit them to the Altair Grid Engine system.

```
# qrsub -a <start_time> -e <end_time>
```

The start and end times are in [[CC]YY]MMDDhhmm[.SS] format. If no start time is given, Altair Grid Engine assumes the current time as the start time. It is also possible to set a duration instead of an end time.

```
# qrsub -a <start_time> -d <duration>
```

The duration is in hhmm[.SS] format. Examples: The following example reserves an slot in the queue all.q in host host1 starting at 04-27 23:59 for 1 hour.

```
# qrsub -q all.q -l h=host2 -a 04272359 -d 1:0:0
```

Many of the options available for qrsub are the same as for qsub.

Monitoring Advance Reservations `qrstat` is the command to list and show all advance reservations known by the Altair Grid Engine system. To list all configured advance reservations type:

```
# qrstat
```

To list a special advance reservation type:

```
# qrstat <ar_id>
```

Every submitted AR has an own ID and a special state.

Table 30: Possible Advance Reservation States

State	Description
w	Waiting - Granted but start time not yet reached
r	Running - Start time reached
d	Deleted - Deleted manually
W	Warning - AR became invalid but start time is not yet reached
E	Error - AR became invalid and start time is reached

Examples:

```
# qrstat
ar-id   name   owner   state  start at            end at              duration
-----------------------------------------------------------------------------
    1          user1   w     04/27/2011 23:59:00 04/28/2011 00:59:00 01:00:00

# qrstat -ar 1
id                           1
owner                        user1
group                        users
name
account                      sge
project
jclass
submission_time              04/27/2011 15:00:11
start_time                   04/27/2011 23:59:00
end_time                     04/28/2011 00:59:00
duration                     01:00:00
free_resources               false
reserve_available_only       false
immediate                    false
standing_reservation         false
submit_cmd                   qrsub -q all.q -l h=host2 -a 04272359 -d 1:0:0
state                        w
resource_list                hostname=host2
granted_slots_list           all.q@host2=1
```

Modifying Advance Reservations `qralter` is the command used to modify already existing advance reservations.

Example:

```
# qrsub -a 201810101200 -e 201810101600
Your advance reservation 123 has been granted
# qralter -e 201810101800 123
modified advance reservation 123
```

All attributes of an advance reservation which can be specified at AR submission time can also be modified, provided that the resource consumption in the cluster allows for the change.

All attributes can be changed for ARs being still pending or being running but without jobs running in the AR. It might become necessary to reschedule the AR, e.g. if resource requests (-l / -masterl) or queue requests (-q / -masterq) are modified. If rescheduling is not possible as the requested resources are not available in the given time frame then qralter will print an error message and the AR will not be modified.

If an AR is already running and has running jobs

- Simple modifications not affecting the reserved resources like modifying the name (-N) or the account string (-A) will always work.
- Modifying start time (-a), end time (-e) or duration (-d) will work, if the resources held by the AR will also be available in the new time frame. Reducing the time frame will always be accepted.
- If rescheduling of the AR would be necessary as e.g. resource requests shall be modified (-l / -masterl) or the given set of resources will not be available in an extended time frame (-e / -d) qralter will print an error message and the AR will not be modified.
- If rescheduling of the AR would be necessary but is prevented by jobs running in the AR it is possible to enable detaching of jobs from the AR by setting the qmaster_param AR_DETACH_JOBS_ON_RESCHEDULE in the global configuration, see also sge_conf(5).

Deleting Advance Reservations `qrdel` is the command to delete an advance reservation. The command requires at least the ID or the name of the AR.

Example:

```
# qrdel 1
```

A job which refers to an advance reservation which is in deletion will also be removed. The AR will not be removed until all referring jobs are finished!

Using Advance Reservations Advance Reservations can be used via the -ar <ar_id> parameter which is available for qsub, qalter, qrsh, qsh and qlogin.

Example:

```
# qsub -ar 1 reservation_job.sh
```

Overwriting the consumable attribute of resources with AR submission In certain scenarios it may be useful or necessary to overwrite the consumable attribute of a requested resource. Imagine the following situation:

- the machines in your cluster have 32 cores and 2 GPUs per node
- GPU is a per job resource (defined with consumable JOB in the complex definition)
- you want to submit an advance reservation that will spawn multiple hosts and run multiple parallel jobs each requesting one GPU and an arbitrary number of cores on one host

An AR submission

```
# qrsub -pe mype 128 -l GPU=2 -d 7200
```

will give you 4 hosts but only 2 GPUs on one (the first) of the hosts

```
# qrsub -pe mype 128 -l GPU=8 -d 7200
```

will not succeed as it would request 8 GPUs on the first host which are not available

You need to tell Altair Grid Engine to allocate 2 GPUs per host, which can be done by overwriting the per JOB definition of the complex variable by per HOST:

```
# qrsub -pe mype 128 -l GPU=2{HOST} -d 7200
```

This will give you 4 hosts with 2 GPUs each. You can then submit multiple jobs in the AR which can get 1 or 2 GPUs per job, e.g.

```
# qsub -ar ar_id -pe mype 16 -l GPU=1
# qsub -ar ar_id -pe mype 64 -l GPU=2
```

7.7.2 Standing Reservations

Standing Reservations can only be created by users which are in the *arusers* list.

In order to create a Standing Reservation a calendar needs to be specified. The calendar determines the start and end times of the Advance Reservations which are dynamically created by the Standing Reservation.

Standing Reservations are per default endless unless an end time is specified either by the duration or by the end time switch.

The first allocated Advance Reservation is the next matching start date of the calendar unless a later start time is specified.

The scheduler allocates per default the next 8 Advance Reservation instances during submission time. Whenever an Advance Reservation ends it allocates one more Advance Reservation to keep the instance count constant. The amount of allocated Advance Reservations at a time is called depth and is a qrsub parameter (`-cal_depth`). The administrator can limit the maximum depth with the MAX_AR_CAL_DEPTH qmaster parameter. Per default it is limited to 8.

In case an Advance Reservation instance cannot be allocated as the required resources are not available at the given time interval it will go into Error (E) state and jobs will not be dispatched into this AR.

If the first Advance Reservation instance cannot be allocated submission of the Standing Reservation will be rejected. The user can override this behaviour with the `-cal_jmp` parameter. The `-cal_jmp` parameter specifies how many non allocatable Advance Reservation instances may be skipped at Standing Reservation submission time without the submission being rejected.

The administrator can limit the amount of non-allocated reservations with the MAX_AR_CAL_JMP parameter. Per default it is limited to 0.

Creating Standing Reservations Standing Reservations can be created with the `qrsub` command using a calendar specification. The calendar specification is the `-cal_week` parameter which accepts a Altair Grid Engine weekly calendar specification (see `man calendar_conf`). The weekly calendar is only allowed to set the state *on*.

```
$ qrsub -cal_week "mon-fri=08:30-11:30=on" -q all.q -pe mytestpe 10
Your advance reservation 4000000000 has been granted
```

This command creates Advance Reservations for each day from Monday till Friday from 8:30 till 11:30 in the queue `all.q` for one slot. As with Advance Reservations, if multiple slots are required, a parallel environment with a certain amount of slots needs to be requested.

The above command tries to create the default amount of Advance Reservations starting from the next possible day. If an Advance Reservation ends a new one is allocated automatically after the last one in the schedule. In order to limit the amount of allocated Advance Reservations at one point in time the `-cal_depth` parameter has to be used. Following command allocates just one Advance Reservation when that one finishes a new one for the next possible date is created.

```
$ qrsub -cal_week "mon-fri=08:30-11:30=on" -cal_depth 1 -q all.q -pe mytestpe 10
Your advance reservation 4000000001 has been granted
```

It is possible that some of the Advance Reservations can not be allocated in the schedule since resources are already in use. If this happens no further Advance Reservations are scheduled. In order to explicitly allow that unallocated times can be skipped the `-cal_jmp` parameter can be used. This parameter determines how many time ranges are allowed to be skipped without an Advance Reservation if there are no resources available. Per default it is 0.

In the following example there is only 1 slot configured in the all.q with one host.

Now 3 Advance Reservations are scheduled with an unlimited calendar.

```
$ qrsub -cal_week "mon-fri=08:30-11:30=on" -cal_depth 3 -q all.q
Your advance reservation 4000000002 has been granted
```

In order to inspect the Standing Reservation the `qrstat` command line tool can be used.

```
$ qrstat -ar 4000000002

id                                  4000000002
owner                               daniel
group                               daniel2
name
account                             sge
project
jclass
submission_time                     03/30/2016 11:34:34.815
start_time                          NONE
```

```
end_time                          NONE
duration                          NONE
free_resources                    false
reserve_available_only            false
immediate                         false
standing_reservation              true
sr_cal_week                       mon-fri=08:30-11:30=on
sr_cal_depth                      8
sr_cal_jmp                        0
submit_cmd                        qrsub -cal_week "mon-fri=08:30-11:30=on"
                                  -cal_depth 3 -q all.q
sr_state_0                        w
sr_start_time_0                   03/31/2016 08:30:00.000
sr_end_time_0                     03/31/2016 11:30:00.000
sr_duration_0                     03:00:00.000
sr_allocated_0                    true
sr_granted_parallel_environment_0
sr_granted_slots_list_0           all.q@mint14=1
sr_state_1                        w
sr_start_time_1                   04/01/2016 08:30:00.000
sr_end_time_1                     04/01/2016 11:30:00.000
sr_duration_1                     03:00:00.000
sr_allocated_1                    true
sr_granted_parallel_environment_1
sr_granted_slots_list_1           all.q@mint14=1
sr_state_2                        w
sr_start_time_2                   04/04/2016 08:30:00.000
sr_end_time_2                     04/04/2016 11:30:00.000
sr_duration_2                     03:00:00.000
sr_allocated_2                    true
sr_granted_parallel_environment_2
sr_granted_slots_list_2           all.q@mint14=1
free_resources                    false
```

The next Standing Reservation is requested with the same calendar. Hence the slots can not be granted.

```
$ qrsub -cal_week "mon-fri=08:30-11:30=on" -cal_depth 3 -q all.q
Could not find time slots for Standing Reservation
```

But the Standing Reservation request can be allowed to skip (jump) over reservation times when there are not enough slots.

```
$ qrsub -cal_week "mon-fri=08:30-11:30=on" -cal_depth 1 -cal_jmp 4 -q all.q
Your advance reservation 4000000003 has been granted
```

When inspecting the Standing Reservation it can be noticed that one Advance Reservation was allocated (*sr_allocated true*) due to a depth of 1 and 3 times of the calendar scheduler are skipped over.

```
$ qrstat -ar 4000000003

id                                4000000003
owner                             daniel
group                             daniel2
name
account                           sge
project
jclass
submission_time                   03/30/2016 11:34:51.778
start_time                        NONE
end_time                          NONE
duration                          NONE
free_resources                    false
reserve_available_only            false
immediate                         false
standing_reservation              true
sr_cal_week                       mon-fri=08:30-11:30=on
sr_cal_depth                      8
sr_cal_jmp                        0
submit_cmd                        qrsub -cal_week "mon-fri=08:30-11:30=on"
                                  -cal_depth 1 -cal_jmp 4 -q all.q
sr_state_0                        w
sr_start_time_0                   03/31/2016 08:30:00.000
sr_end_time_0                     03/31/2016 11:30:00.000
sr_duration_0                     03:00:00.000
sr_allocated_0                    false
sr_granted_parallel_environment_0
sr_granted_slots_list_0           all.q@mint14=1
sr_state_1                        w
sr_start_time_1                   04/01/2016 08:30:00.000
sr_end_time_1                     04/01/2016 11:30:00.000
sr_duration_1                     03:00:00.000
sr_allocated_1                    false
sr_granted_parallel_environment_1
sr_granted_slots_list_1           all.q@mint14=1
sr_state_2                        w
sr_start_time_2                   04/04/2016 08:30:00.000
sr_end_time_2                     04/04/2016 11:30:00.000
sr_duration_2                     03:00:00.000
sr_allocated_2                    false
sr_granted_parallel_environment_2
sr_granted_slots_list_2           all.q@mint14=1
sr_state_3                        w
sr_start_time_3                   04/05/2016 08:30:00.000
sr_end_time_3                     04/05/2016 11:30:00.000
sr_duration_3                     03:00:00.000
sr_allocated_3                    true
sr_granted_parallel_environment_3
```

```
sr_granted_slots_list_3            all.q@mint14=1
```

Creating a Time Restricted Standing Reservation In all previous examples there is no start date for the first nor an end date for the last Advance Reservation. By using the `-a`, `-d`, and `-e` switches of qrsub together with a calendar request, the Advance Reservations scheduled within Standing Reservations can be limited.

The `-a` switch denotes the start time of the Standing Reservation. It is the earliest time the first AR within the Standing Reservation can be scheduled. Unlike for Advance Reservations it is valid to specify a start time without an end time or duration. If an end time is specified with `-e` then the last scheduled AR must end before the given time. Like the `-a` switch also the `-e` switch can requested as a single request.

The duration switch `-d` can be requested either with a start time (`-a`) which then specifies the end time or without any other request which then specifies the end time as the current time plus the duration. The next example demonstrates how a Standing Reservation with ARs scheduled only in the first week of April 2016 can be requested. Note that `-a`, `-e` are *date_time* requests ([[CC]YY]MMDDhhmm[.SS]) while the `-d` parameter is a time request (hours:minutes:seconds or seconds).

```
$ qrsub -cal_week "mon-fri=08:30-11:30=on" -a 1604010000 -e 1604080000 -q all.q
Your advance reservation 4000000005 has been granted

$ qrstat -ar 4000000005

id                                 4000000005
owner                              daniel
group                              daniel2
name
account                            sge
project
jclass
submission_time                    03/30/2016 13:12:19.970
start_time                         04/01/2016 00:00:00.000
end_time                           04/08/2016 00:00:00.000
duration                           168:00:00.000
free_resources                     false
reserve_available_only             false
immediate                          false
standing_reservation               true
sr_cal_week                        mon-fri=08:30-11:30=on
sr_cal_depth                       8
sr_cal_jmp                         0
submit_cmd                         qrsub -cal_week "mon-fri=08:30-11:30=on"
                                   -a 1604010000 -e 1604080000 -q all.q
sr_state_0                         w
sr_start_time_0                    04/01/2016 08:30:00.000
sr_end_time_0                      04/01/2016 11:30:00.000
sr_duration_0                      03:00:00.000
```

```
sr_allocated_0                     true
sr_granted_parallel_environment_0
sr_granted_slots_list_0            all.q@mint14=1
sr_state_1                         w
sr_start_time_1                    04/04/2016 08:30:00.000
sr_end_time_1                      04/04/2016 11:30:00.000
sr_duration_1                      03:00:00.000
sr_allocated_1                     true
sr_granted_parallel_environment_1
sr_granted_slots_list_1            all.q@mint14=1
sr_state_2                         w
sr_start_time_2                    04/05/2016 08:30:00.000
sr_end_time_2                      04/05/2016 11:30:00.000
sr_duration_2                      03:00:00.000
sr_allocated_2                     true
sr_granted_parallel_environment_2
sr_granted_slots_list_2            all.q@mint14=1
sr_state_3                         w
sr_start_time_3                    04/06/2016 08:30:00.000
sr_end_time_3                      04/06/2016 11:30:00.000
sr_duration_3                      03:00:00.000
sr_allocated_3                     true
sr_granted_parallel_environment_3
sr_granted_slots_list_3            all.q@mint14=1
sr_state_4                         w
sr_start_time_4                    04/07/2016 08:30:00.000
sr_end_time_4                      04/07/2016 11:30:00.000
sr_duration_4                      03:00:00.000
sr_allocated_4                     true
sr_granted_parallel_environment_4
sr_granted_slots_list_4            all.q@mint14=1
```

Submitting Jobs into Standing Reservations After a Standing Reservation was granted the given ID can be used like a Advance Reservation ID, i.e. the `qsub -ar <ID>` parameter needs to be used to submit jobs into the Advance Reservations given by Standing Reservation. When one Advance Reservation ends jobs running in the Advance Reservation are deleted. Jobs which are still queued remain waiting for the next occurrence of an Advance Reservation instance of the Standing Reservation. By using the job start time parameter `-a` jobs can be further directed not to start in any Advance Reservation of the Standing Reservation before that time. When a Standing Reservation ends (when having an end time specified or no further Advance Reservations can be allocated within the given constraints) all jobs, also waiting jobs are deleted.

The following example shows how to request a Standing Reservation.

```
$ qsub -ar 4000000000 myjob.sh
```

Monitoring Standing Reservations In order to display the individual Advance Reservation instances of a Standing Reservation the qrstat command line utility can be used.

Standard Advance Reservations and Standing Reservations are differentiated by the *standing_reservation* entry (for `qrstat -ar`) or the *sr* (Standing Reservation) column (for `qrstat`).

In the qrstat overview the state refers to the state of the next AR within the Standing Reservation. The start at, end at, as well as duration column refer to the start, end, and duration of the whole Standing Reservation. If the Standing Reservation is not limited NONE is shown.

Following an example in which the first 3 entries are Standing Reservation and the last entry is an Advance Reservation.

```
$ qrstat
ar-id      name owner        state start at            end at              duration  sr
-------------------------------------------------------------------------------------------
4000000013      daniel       w     NONE                NONE                NONE      true
4000000015      daniel       r     NONE                NONE                NONE      true
4000000017      daniel       w     03/30/2016 13:52:32 04/01/2016 13:52:32 48:00:00  true
4000000019      daniel       w     10/10/2018 00:00:00 10/15/2018 08:00:00 128:00:00 false
```

Detailed information about the scheduled instances of the Advance Reservations within a Standing Reservation can be requested with the `qrstat -ar` switch. Following entries can be seen:

Table 31: TABLE: Standing Reservation details

Entry	Specification
`sr_cal_week`	Shows the cal_week submission request.
`sr_state_0`	Shows the state of the first instance within the Standing Reservation.
`sr_start_time_0`	Shows the start time of the first instance within the Standing Reservation.
`sr_end_time_0`	Shows the end time of the first instance within the Standing Reservation.
`sr_duration_0`	Shows the duration of the first instance within the Standing Reservation.
`sr_allocated_0`	Shows if the first instance could reserve the required resources or not. If that is set to false then the `-cal_jmp` parameter must be set to > 0. Note that in unallocated instance no jobs can run since no resources are free. The allocation is only tried once, for the first (amount given by the `-cal_depth` parameter) ARs during `qrsub` time and for later ARs whenever one AR within the Standing Reservation ends.
`sr_granted_parallel_environment_0`	Optionally shows the parallel environment the first instance within the Standing Reservation got granted.
`sr_granted_slots_list_0`	Shows the queue instances and the number of slots per queue instance the first instance within the Standing Reservation got granted.

For each scheduled Advance Reservation within the Standing Reservation a similar block of entries is shown with the corresponding AR instance number.

The following example shows the details of a Standing Reservation with 2 allocations (depth) which continues to allocated more ARs until it is explicitly deleted (`qrdel`) or no resources can be found for further scheduling more AR instances.

```
$ qrstat -ar 4000000003
--------------------------------------------------------------------------------
id                              4000000003
owner                           daniel
group                           daniel2
name
account                         sge
project
jclass
submission_time                 04/08/2016 11:25:43.137
start_time                      NONE
end_time                        NONE
duration                        NONE
free_resources                  false
reserve_available_only          false
immediate                       false
standing_reservation            true
sr_cal_week                     8:30-11:30=on
sr_cal_depth                    2
sr_cal_jmp                      0
submit_cmd                      qrsub -cal_week "mon-fri=08:30-11:30=on"
                                -a 1604010000 -e 1604080000 -q access
                                -cal_depth 2
sr_state_0                      w
sr_start_time_0                 04/09/2016 08:30:00.000
sr_end_time_0                   04/09/2016 11:30:00.000
sr_duration_0                   03:00:00.000
sr_allocated_0                  true
sr_granted_parallel_environment_0
sr_granted_slots_list_0         access@u1010=1
sr_state_1                      w
sr_start_time_1                 04/10/2016 08:30:00.000
sr_end_time_1                   04/10/2016 11:30:00.000
sr_duration_1                   03:00:00.000
sr_allocated_1                  true
sr_granted_parallel_environment_1
sr_granted_slots_list_1         access@u1010=1
```

7.8 Jobs using Docker Containers

Docker containers allow to run applications with specific demands for their software environment without the need to keep separate hosts just to provide that specific environment.

Docker containers are - from the users point of view - similar to virtual machines, but are much lighter and simpler and are easier to maintain.

Altair Grid Engine provides an integration with Docker which allows to start tasks of jobs inside Docker containers. Currently, this integration is supported only on newer Linux versions. If Docker is installed on an execution host, this reports both the availability of Docker on this host and the list of available Docker images. The availability of Docker is reported by the `docker` complex of type `BOOL`; if Docker is available, the value is `true`. The list of locally available Docker images on this host is reported as a comma separated list in the complex `docker_images`, which is of type `RESTRING`. The comma separated list has the format `REPOSITORY:TAG[,REPOSITORY:TAG,...]`, where the `REPOSITORY` and `TAG` define the Docker image like in the output of the `docker images` command. Because all images are reported as one string, the request for an image must select one part of this string, which is done by using wildcards, see the examples below.

7.8.1 Running a sequential job in a Docker container

There are two ways to start a sequential job in a Docker container:
1) Specify the job or job script to start on the job submit command line
2) Let Docker start whatever is defined as the `ENTRYPOINT` of the container

For both ways, to submit a job to a Docker container both the `docker` and the `docker_images` complex must be requested, like in this job submit of the first kind:
`$ qsub -l docker,docker_images="*centos:latest*" my_job.sh`

This job will be started
a) on a host with a running and properly answering Docker daemon of at least version 1.8.3 (see the AdminGuide or the Release Notes for the latest supported Docker version)
b) where a Docker image is available that matches `"*centos:latest*"`

Because this is not a `binary` job, the job script gets transferred from the submit host to the execution host by Altair Grid Engine. There, the script is copied to the job spool directory of the execution daemon. In order to allow this script to be started inside of the container, the spool directory must be made available inside the container (which is called "binding" in the Docker terminology). Also, the $SGE_ROOT must always be available inside the container to allow Altair Grid Engine to work properly. Furthermore, Altair Grid Engine automatically detects other directories that have to be available inside the container in order to allow the job to run.

Sharing and binding these directories is done automatically by Altair Grid Engine. These directories are always bound to a subdirectory of `/uge_mnt` inside the container, and they are bound by sharing each the top level directory to a direct subdirectory of the '/uge_mnt' bind point with the same name. I.e.:

- If $SGE_ROOT is e.g. `/opt/uge`, then the top directory `/opt` is bound to `/uge_mnt/opt` inside the container.
- If the execution daemon spool directory is e.g. `/var/spool/uge`, then the top directory `/var` is bound to `/uge_mnt/var` inside the container.

The top level directory and not the specific directory itself is bound because Docker was not able to bind a directory to a bound directory in older versions, which would be the case

if both `/opt:/uge_mnt/opt` and `/opt/uge:/uge_mnt/opt/uge` would be bound automatically - then the `uge` subdirectory would be bound to the already bound `/uge_mnt/opt` directory, which wasn't allowed. This behaviour could be changed in future versions of UGE.

There are other directories that are bound automatically inside the container, e.g. the job users home directory to allow the output and error file of the job to written to their default location. If the paths specified with the `-o` and `-e` switch point to different directories, these are bound into the container instead. This automatic directory binding applies to all directories that are defined explicitly or implicitly by specifying or omitting Altair Grid Engine switches.

Altair Grid Engine cannot detect which paths the job itself uses, even if they are specified as arguments to the job script. For this, paths must be bound manually, using the `-xd -v` switch, which takes the argument `HOST-DIR:CONTAINER-DIR` (see the submit(1) man page for details).

Docker disallows to bind two different directories to the same bind point inside the container. Among all the automatic and manual directory binds, Altair Grid Engine ensures a directory is not bound two times to the same top directory inside of the container. The user does not have to take care of this.

Paths that are automatically bound by Altair Grid Engine are also automatically mapped, i.e. the Altair Grid Engine components running inside the container use the bound paths instead of the original ones. But a path in an argument to the job cannot be mapped automatically, this must be done by the job submitter or the job script. E.g. if a job is submitted using this command line:

```
$ qsub -l docker,docker_images="*centos:latest*" -xd "-v /scratch:/container_scratch"
my_job.sh /scratch/data
```

it will not work if the `my_job.sh` script does not map the data path to `/container_scratch/data` or the submit command line is changed to specify `/container_scratch/data` as the job argument.

Furthermore, the job user will probably not exists inside the container. While the whole container is started under the job users ID, the job user itself is not configured inside the container, so the home directory cannot be estimated and must be set explicitly.

If the job is `binary`, i.e. was submitted with the `-b y` switch, the binary is started in the `shell` that is defined in the configuration of the queue the job runs in. Because this is `/bin/csh` by default and the csh is not part of most Docker images, this shell must be overwritten by the `-S` switch - `/bin/sh` usually exists everywhere.

Here are some examples for jobs that use the Docker integration and specify the job binary or job script to start:

- `$ qsub -l docker,docker_images="*centos:latest*" -xd "-v /scratch:/data" my_job.sh /data/input.txt`
  This job requests to be started in a Docker container that is created from the "centos:latest" image, the `/scratch` directory to be bound inside the container to the `/data` directory. The job script is transferred from the submit host to the execution host, the argument to the job script uses the path as it will be called inside of the container.

- `$ qsub -l docker,docker_images="*ubuntu:14.04*" -b y -S /bin/sh hostname`
  This job requests to be started in a Docker container that is created from the "ubuntu:14.04" image, it is a `binary` job which means the job binary or script already exists on the execution host *inside the container*. Because the binary would be started inside the shell configured in the queue, which does not exist in the container, the Bourne shell is defined to be used instead.
- `$ qrsh -l docker,docker_images="*centos:7*" ls -la /uge_mnt`
  This job is an interactive job, it requests to be started in a Docker container that is created from the "centos:7" image. By default, an interactive job is a binary job, i.e. the job script or binary is expected to already exist on the execution host *inside the container*. It lists the automatically bound directories.

In order to submit a job that starts the Docker container itself like a binary, i.e. uses the `ENTRYPOINT` defined in the Docker image instead of a job script, the keyword `NONE` must be used as job script and the job must be a `binary` job, e.g.:

```
$ qsub -l docker,docker_images="*hello-world:latest*" -b y NONE arg1 arg2 argn
```

These jobs are called "autostart Docker jobs". For such jobs, the following limitations apply:

- Only sequential batch jobs are supported, but neither interactive jobs submitted by `qrsh` or `qlogin` nor parallel jobs are supported.
- Stdin cannot be redirected to the job.
- The job can only be suspended, unsuspended and killed, but no other signals can be sent to the job.
- If the Docker daemon is stopped or dies, Altair Grid Engine has no means to control the job.
- In principle it is possible to provide arguments to this kind of jobs, but the arguments will be available to the script or binary inside the Docker container only if the Docker image was created in a way that allows this. Whether the Docker image is suitable can be tested by manually starting
  `$ docker run -it image:latest arg1 arg2 arg3`
  on the execution host. If the script or binary inside the Docker container gets these arguments, it should also get them in a Altair Grid Engine job. If there are arguments specified in the `ENTRYPOINT` of the Docker image, the arguments specified on the command line will be appended to them.

What happens under the hood Altair Grid Engine directly communicates with the Docker daemon using the Docker Remote API and does not use the `docker` command line client. The Remote API is a stateless request-response interface, similar to a web server.

If Altair Grid Engine decides to start a job on a certain execution host in a certain Docker container, it fills requests forms with information and sends them to the Docker daemon. The Docker daemon tries to fulfill each request and responds to each request. Each response contains a status (success or failure) and some responses additionally contain data or an error message.

To start a normal Docker job, Altair Grid Engine sends these requests to the Docker daemon:

- A request to create a Docker container from the specified Docker image, with the start user being the job user, the start application being the `sge_container_shepherd`, the path bindings and so on.
- A request to give the Docker container its name containing the job ID.
- A request to start the Docker container.

The Docker daemon does this to fulfill the requests:

- To create a Docker container, it extracts the Docker image to a subdirectory and writes the specified information to a specific file.
- The container name is changed in the internal database of the Docker daemon.
- To start the Docker container, the Docker daemon sets up the environment, defines the extracted directory to be the root directory for the process to start and executes the `sge_container_shepherd`, which acts as init process of the container, i.e. there is no extra process which "is" the container - the container "is" the application that gets started, which is the `sge_container_shepherd` for this kind of jobs.

The `sge_container_shepherd` then starts the job with all of its arguments, exactly like the normal `sge_shepherd` does with normal jobs on the "real" host. If a signal is to be sent to the job, the execution daemon talks to the `sge_container_shepherd` via a pipe. The `sge_container_shepherd` sends the signal to the job.

If Altair Grid Engine decides to start a Docker job that uses the keyword "NONE" as job script, it does the same as above, but the start application is the one defined in the Docker image and is not explicitly set. If the container is created, there is no Altair Grid Engine component running in it, so Altair Grid Engine has no direct control over the container. Instead, it must send a request to the Docker daemon in order to send signals to the job, in order to get online usage, and so on.

7.8.2 Running a parallel Job in Docker containers

For tightly integrated parallel jobs, all tasks except for the master task are started in separate Docker containers that are created from the same Docker image. For loosely integrated parallel jobs, no task is started in a Docker container by Altair Grid Engine, because Altair Grid Engine has no control over the slave tasks, i.e. submitting loosely integrated parallel Docker jobs does not make sense.

Like any normal parallel job, the master task is started like a sequential job that requests a parallel environment with an amount of slots. Parallel Docker jobs additionally have to provide the Docker specific requests:

```
$ qsub -l docker,docker_images="*centos:latest*" -xd "-v /home:home" -l /home/jdoe
-j y      -pe parallel_env 3 master_job.sh
```

The master task is submitted to a "physical" host which is known to Altair Grid Engine but it is started inside a Docker container, which has a random name.

The slave tasks are just submitted using the usual `-inherit` switch which requires two environment variables set in the submit shell:

```
$ export JOB_ID=17
```

```
$ export SGE_TASK_ID=undefined
$ qrsh -inherit slave_host slave_job.sh
```

All Docker specific request are inherited from the master task and may not be provided in the qrsh command line!

The slave task is submitted to the `slave_host`, which is a “physical” host and is known to Altair Grid Engine. The task itself then is started inside a Docker container which has a random name and is not known to Altair Grid Engine.

Usually the master task of that parallel job runs the `qrsh -inherit`, which would not work for a master task running inside a Docker container, because this Docker container is not known as an execution host to the `sge_qmaster` and the execution daemons on the execution hosts.

To solve this problem, the Administrator can declare a RSMAP complex that defines on each execution host as many container hostnames as there are slots defined. The job then must request one element per PE task of this RSMAP complex and the `per_pe_task_prolog` must be used to set the hostname and IP of the Docker container.

E.g.: * the Administrator declares an RSMAP complex “cont_hosts” * on execution host “hostA”, defines the complex values “cont_hosts=2(hostA_cont1 hostA_cont2) * on execution host”hostB“, defines the complex values”cont_hosts=2(hostB_cont1 hostB_cont2) * the job has to request one element of this complex per task: `$ qsub -l cont_hosts=1,docker,docker_images="*centos:latest*" -xd "-v /home:home" -l /home/jdoe          -j y -pe parallel_env 3 master_job.sh` Assume the master task is scheduled to “hostA”, both slave tasks are scheduled to “hostB”. The master task gets the RSMAP value “hostA_cont2” assigned, the slave tasks get “hostB_cont1” and “hostB_cont2” assigned, so the “per_pe_task_prolog” can set the hostnames and IPs accordingly. With having the hostnames and IPs registered in DNS, all components now can talk to each other.

7.8.3 Running MPI jobs in Docker containers

If MPI is used for the communication between the master and slave tasks, a file that contains the hostnames can be written automatically be Altair Grid Engine. If the “execd_params” value “CONTAINER_PE_HOSTFILE_COMPLEX” is set to the name of the RSMAP complex used to select the container hostnames, Altair Grid Engine automatically writes a “container_pe_hostfile” which is compatible to the normal “pe_hostfile”, but contains all container names selected for this job. The “prolog” can be used to replace the normal “pe_hostfile” with the “container_pe_hostfile” in case this shall be used.

7.8.4 Running an array Job in Docker containers

Each task of an array job is started in a different Docker container, but all Docker containers are created from the same Docker image.

7.8.5 Running a Job in a Docker image that is not available locally

Docker allows not only to use locally available images, but also to automatically download images from a repository. Because of performance considerations, this is sometimes not wanted for Altair Grid Engine jobs, so usually a job is scheduled only to an execution host that already provides the requested Docker image. If a job has to run in an image that is not yet available, submitting it with a soft request for that image triggers the download of this image. This means, for a job like this one:

```
$ qsub -l docker -soft -l docker_images="*fedora:21*" -o /dev/null -j y myjob.sh
```

Altair Grid Engine will first search for a execution host that fulfills all requests, i.e. that already has the Docker image `fedora:21` locally available. If there is no such host in the cluster, the job will be scheduled to any execution host that fulfills the `docker` request and will tell the Docker daemon to download the image and the start the container.

7.8.6 Using placeholders to dynamically define Docker options

Since Altair Grid Engine 8.5.0, placeholders are allowed in sub-options of the "-xd" option on the submit command line, sge_request files, job scripts, job classes and job submission verifier. These placeholders are resolved by corresponding elements of specific RSMAP complexes the Scheduler selects for the tasks of a job.

The format of these placeholders is:

```
<placeholder> := ${ <complex_name> "(" <index> ")" }
```

where complex_name is the name of the corresponding RSMAP complex and index is the index of the element the Scheduler selects from the RSMAP for this job, starting with 0.

E.g.:
If a resource map defines these values on a host: `gpu_map=4(0 1 2 3)`
this qsub command line is used:
(Note the "\$" to keep the shell from trying to resolve that variable)

```
# qsub -l docker,docker_images="*some_image*",gpu_map=2
       -xd "--device=/dev/gpu\${gpu_map(0)}:/dev/gpu0,
            --device=/dev/gpu\${gpu_map(1)}:/dev/gpu1" ...
```

and the scheduler selects the elements "1" and "3" from the resource map, the command line is resolved to

```
# qsub -l docker,docker_images"*some_image*",gpu_map=2
       -xd "--device=/dev/gpu1:/dev/gpu0,
            --device=/dev/gpu3:/dev/gpu1"...
```

which means the physical GPUs "gpu1" and "gpu3" are mapped to the virtual GPUs "gpu0" and "gpu1" inside the container and at the same time are exclusively reserved for the current job among all Altair Grid Engine jobs.

7.8.7 Support for nvidia-docker 2.0

NVIDIA provides the version 2.0 of their Docker Container Runtime which allows to access GPUs from within Docker containers. Altair Grid Engine now supports using this Container Runtime.

Provided the NVIDIA Docker Container Runtime is installed properly on an execution host, a job that wants to use a NVIDIA GPU must tell Docker to use the NVIDIA Runtime by specifying the `-xd "--runtime=nvidia"` switch on the **qsub** or **qrsh** command line. In order to select a specific GPU, the environment variable `NVIDIA_VISIBLE_DEVICES` must be set to for the whole container by specifying it with the `-xd "--env NVIDIA_VISIBLE_DEVICES=0"` switch.

Altair Grid Engine also supports the Docker run option `gpus` to select GPUs for the container. The switch accepts either `all` to select all GPUs on the host, any integer > 0 to select a specific amount of GPUs or the parameter `device` followed by a list of device ids to select specific GPUs, e.g. `-xd "--gpus=device=\"0,1\""`. Please note that `-xd "--gpus=..."` requires Docker API version 1.40 or newer.

8 Getting a Consistent View onto the System by Using Sessions

When Altair Grid Engine client commands interact with Altair Grid Engine server components then this is done by using an interface named GDI (Grid Engine Data Interface). This interface is used to send client requests to the Altair Grid Engine system that are then handled within the server component and answered by a response message that contains the result for the client request.

This GDI interface is also used for internal Altair Grid Engine communication between components running on execution hosts as well as for internal communication between components within the `sge_master` component itself.

GDI requests can be divided into two categories: Requests that will change the configuration/state of the Altair Grid Engine system (read-write-requests) and requests that will gather information to display the configuration/state of the Altair Grid Engine system (read-only-requests).

Altair Grid Engine 8.2 has been redesigned so that read-write-requests and read-only-requests can be executed completely independently from each other. Furthermore up to 64 read-only requests can work in parallel which is not possible in Sun Grid Engine, Oracle Grid Engine and other open source versions of Grid Engine. This ensures faster response times for all requests and has a huge positive impact on the cluster throughput.

The drawback of this approach is that GDI read-only-requests might not see the outcome of recently executed read-write requests in certain situations. E.g. it might happen that a user submits a job (read-write-request) and immediately does a qstat -j (read-only-request) which responds with an error which says that the previously created job does not exist.

In some cases such behavior may cause problems and it is desired that requests should be executed in sequence and for this reason GDI sessions have been introduced that guarantee a consistent view onto the Altair Grid Engine system. Internally read-only requests that

are executed within the control of a session are delayed until they can see all changes that have happened previously.

8.1 Communication with Altair Grid Engine without using Sessions

Altair Grid Engine can be installed in a way so that no sessions are required to get a consistent view onto the Altair Grid Engine system. In that mode the `sge_qmaster` process of Altair Grid Engine 8.2 behaves the same way as in prior versions. All commands are executed in the same sequence as they are received by `sge_qmaster` and during processing of each of those requests all previous activities are immediately visible without the need to use sessions.

To find out if `sge_qmaster` is running in this mode execute following command:

```
> qconf -stl
reader000
reader001
reader002
reader003
reader004
...
```

The output of the `qconf -stl` command will show the active threads in the `sge_qmaster` process. If there are reader-threads active then sessions are required. If there is no line in the output that starts with `reader` then sessions are not required.

8.2 Using sessions to communicate with the system

Sessions are configuration objects available since Altair Grid Engine 8.2. They are required to get a consistent view onto the Altair Grid Engine when read-only-threads where activated during the installation of the `sge_qmaster` process. The use of sessions might slow down processes within `sge_qmaster` slightly therefore sessions can only be created, modified and deleted by managers or users that are members of the `sessionusers` access control list.

Following session related commands are available:

Table 32: TABLE: Session Commands

Command	Value Specification
`qconf -ssil`	Shows all active sessions including ownership and end time.
`qconf -ssi <session_id>`	Shows details of an existing session object.
`qconf -msi <session_id>`	Opens an editor and lets the user configure the session.
`qconf -Msi <session_file>`	Modifies the session using new parameters from `session_file`
`qconf -asi`	Adds a new session object

Command	Value Specification
qconf -Asi <session_file>	Adds a new session using parameter values from session_file
qconf -csi	Creates a new session with default parameters.
qconf -dsi <session_id>	Deletes the session with the given session_id.

The following list of parameters specifies the session configuration:

Table 33: TABLE: Session Parameters

Parameter	Value Specification
session_id	The session ID of a session. For sessions that should be created the value for this attribute has to be NONE so that the sge_qmaster process can assign a new unique session ID.
owner	User name of the user that owns the session. If NONE is specified as username during the creation of a new session then the executing user of the configuration command will be the owner of that session. Only managers and the session owner are allowed to modify or to delete an existing session and if a session gets created by root or a manager account on behalf of a regular user then that user should be a member of the sessionusers access control list.
duration	The duration influences the lifetime of a session. Lifetime of a session begins when the session is created and it ends when the session is not used for the specified amount of time defined by the duration attribute. Lifetime of a session is automatically increased by adding duration to the end_time of that session when it is used. The default duration of a session is 900 seconds if this is not specified otherwise in the qmaster_param namedgdi_request_session_timeout'. The sge_qmaster process tries to find sessions where the lifetime ended every 15 minutes and it will delete those sessions automatically. Although unused sessions will be deleted automatically it is recommended to delete sessions manually using the qconf -dsi command once a session is not needed anymore.
start_time	Time when the session was created. Start_time of a session cannot be specified. It is shown with qconf -ssi.
end_time	Possible end time of a session. After creation the end_time of a session is set to start_time plus duration. End_time is moved forward when the session is used so that it still remains valid for the amount of time specified by duration after use. If the session was not used then it is tagged for deletion. The sge_qmaster process tries to find unused sessions every 15 minutes and it will delete those sessions automatically. Although unused sessions will be deleted automatically it is recommended to delete sessions manually using the qconf -dsi command when a session is not needed anymore.

Parameter	Value Specification
	The end_time of a session is shown by the commands qconf -ssi and -ssil.

Sessions can be used with the -si switch of all client commands (like qsub, qstat, qhost ...). Requests sent by the corresponding client to the sge_qmaster daemon will be done as part of the specified session. If the switch is omitted or if NONE is specified as session_id then such requests will be executed outside the control of a session.

Here is an example that shows the use of a session:

```
> set session_id=`qconf -csi`
> set job_id=`qsub -terse -si $session_id -b y sleep 120`
> qstat -si $session_id -j $job_id
> qconf -dsi $session_id
```

During job submission a session that was previously created is specified. Due to this it is guaranteed that the qstat command that refers to the same session is able to see the previously created job. After use the session is deleted.

9 Submission, Monitoring and Control via an API

Note

Using the API is not supported on Windows hosts.

9.1 The Distributed Resource Management Application API (DRMAA)

The Distributed Resource Management Application API is the industry-leading open standard of the Open Grid Forum www.ogf.org DRMAA working group www.drmaa.org for accessing DRMS. The goal of the API is to provide an external interface to applications for basic tasks, like job submission, job monitoring and job control. Since this standard is adapted by most DRMS vendors it offers a very high investment protection, when developing a DRM aware software application, because it can be easily transferred to another DRM. Altair Grid Engine supports all DRMAA concepts, which allows for the movement of existing DRMAA applications from different DRM vendors.

9.2 Basic DRMAA Concepts

DRMAA version 1.0 specifies a set of functions and concepts. Each DRMAA application must contain an initialization and disengagement function which must be called at the beginning and at the end respectively. In order to do something useful a new DRMAA session must be created or one existing must be re-opened. When re-opening a DRMAA session, the job IDs

of the session can be reused in order to obtain the job status and gain job control. In order to submit jobs, a standard job template must be allocated and filled out according to needs with the job name and the corresponding parameters. This job template than can then be submitted with a job submission routine. There are two job submission routines specified: One for individual jobs and one for array jobs. A job can be monitored and controlled (e.g. holding, releasing, suspending, resuming) once the job is complete and the exit status can be checked. Additionally DRMAA specifies a set of error codes. In order to exploit additional functionality, which is only available in Altair Grid Engine, the standard will allow this with either the native specification functionality or with job categories.

9.3 Supported DRMAA Versions and Language Bindings

Altair Grid Engine supports currently the DRMAA v1.0 standard and is shipped with a fully featured DRMAA C binding v1.0 and a DRMAA Java binding v1.0. The standards can be downloaded at `www.drmaa.org`.

9.4 When to Use DRMAA

Writing applications with DRMAA has several advantages: High job submission throughput with Altair Grid Engine, the defined workflow is independent from underlying DRM, it is much easier to use in programming languages like C or Java, and it is a widely known and adapted standard backed by an experienced community.

9.5 Environment Variable Influences

There are environment variables that can change DRMAA library behavior:

SGE_DRMAA_ENABLE_ERROR_STATE

When this environment variable is set, then jobs that are submitted with drmaa_run_job() or drmaa_run_bulk_jobs() will change into error state when either during the job start or during the execution of the job an error occurs. Normally DRMAA jobs will not switch into error state when something fails.

9.6 Examples

9.6.1 Building a DRMAA Application with C**

Compiling, Linking and Running the C Code DRMAA Example

In order to compile a DRMAA application, the drmaa.h must include the file and the DRMAA library must be available. The drmaa.h file can be found in the `$SGE_ROOT/include` directory and the libraries are installed in `$SGE_ROOT/lib/$ARCH`.

In the following example the root installation directory ($SGE_ROOT) is /opt/uge870 and the architecture is lx-amd64.

```
> gcc -I/opt/uge870/include -L/opt/uge870/lib/lx-amd64 -o yourdrmaaapp yourdrmaaapp.c -ldrmaa
```

In order to run yourdrmaaapp the Altair Grid Engine environment must be present and the path to the shared DRMAA library must be set.

```
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/uge870/lib/lx-amd64
> ./yourdrmaaapp
```

Job Submission, Waiting and Getting the Exit Status of the Job

In the following example a job session is initially opened with drmaa_init(). The return code of the all calls indicate the success of a function (DRMAA_ERRNO_SUCCESS) or if an error has occurred. In the case of an error, the error string with the corresponding message is returned. In order to submit a job, a job template must be allocated with drmaa_allocate_job_template() and the DRMAA_REMOTE_COMMAND parameters must be set. After a successful job submission with drmaa_run_job() the application waits until the job is scheduled and eventually finished. Then the exit code of the job is accessed and printed before the job session is closed by drmaa_exit.

```
000     #include <stdio.h>
001     #include "drmaa.h"
002
003     int main(int argc, char **argv) {
004
005     /* err contains the return code of the called functions */
006     int err = 0;
007
008     /* allocate a string with the DRMAA string buffer length */
009     char errorstr[DRMAA_ERROR_STRING_BUFFER];
010
011     /* allocate a buffer for the job name */
012     char jobid[DRMAA_JOBNAME_BUFFER];
013
014     /* pointer to a job template */
015     drmaa_job_template_t *job_template = NULL;
016
017     /* DRMAA status of a job */
018     int status = 0;
019
020     /* if job exited normally */
021     int exited = 0;
022
023     /* exit code of the job */
024     int exitstatus = 0;
025
026     /* create a new DRMAA session */
027     err = drmaa_init(NULL, errorstr, DRMAA_ERROR_STRING_BUFFER);
028
029     /* test if the DRMAA session could be opened  */
030     if (err != DRMAA_ERRNO_SUCCESS) {
031          printf("Unable to create a new DRMAA session: %s\n", errorstr);
```

```
032         return err;
033     }
034
035     /* allocate a job template */
036     err = drmaa_allocate_job_template(&job_template, errorstr,
037                                       DRMAA_ERROR_STRING_BUFFER);
038
039     /* test if the DRMAA job template could be allocated */
040     if (err != DRMAA_ERRNO_SUCCESS) {
041          printf("Unable to allocate a new job template: %s\n", errorstr);
042          /* close the DRMAA session and exit */
043          err = drmaa_exit(errorstr, DRMAA_ERROR_STRING_BUFFER);
044          if (err != DRMAA_ERRNO_SUCCESS) {
045             printf("Unable to close DRMAA session: %s\n", errorstr);
046          }
047          return err;
048      }
050
051     /* specify the job */
052     err = drmaa_set_attribute(job_template, DRMAA_REMOTE_COMMAND, "./job.sh",
053                               errorstr, DRMAA_ERROR_STRING_BUFFER);
054
055     if (err != DRMAA_ERRNO_SUCCESS) {
056          printf("Unable to set the remote command name: %s\n", errorstr);
057          /* close the DRMAA session and exit */
058          err = drmaa_exit(errorstr, DRMAA_ERROR_STRING_BUFFER);
059          if (err != DRMAA_ERRNO_SUCCESS) {
060             printf("Unable to close DRMAA session: %s\n", errorstr);
061          }
062          return err;
063     }
064
065     /* submit the job */
066     err = drmaa_run_job(jobid, DRMAA_JOBNAME_BUFFER, job_template, errorstr,
067                         DRMAA_ERROR_STRING_BUFFER);
068
069     /* wait for the job */
070     err = drmaa_wait(jobid, NULL, 0, &status, DRMAA_TIMEOUT_WAIT_FOREVER,
071                      NULL, errorstr, DRMAA_ERROR_STRING_BUFFER);
072
073     if (err != DRMAA_ERRNO_SUCCESS) {
074          printf("Unable to wait for the job: %s\n", errorstr);
075          /* close the DRMAA session and exit */
076          err = drmaa_exit(errorstr, DRMAA_ERROR_STRING_BUFFER);
077          if (err != DRMAA_ERRNO_SUCCESS) {
078             printf("Unable to close DRMAA session: %s\n", errorstr);
079          }
080          return err;
081     }
```

```
082
083     /* print the exit status of the job if terminated normally (and don't
084      * check a function error) */
085     drmaa_wifexited(&exited, status, NULL, 0);
086
087     if (exited == 1) {
088        drmaa_wexitstatus(&exitstatus, status, NULL, 0);
089        printf("Exit status of the submitted job: %d\n", exitstatus);
090     }
091
092     /* free the job template */
093     err = drmaa_delete_job_template(job_template, errorstr, DRMAA_ERROR_STRING_BUFFER);
094
095     if (err != DRMAA_ERRNO_SUCCESS) {
096        printf("Unable to delete the job template: %s\n", errorstr);
097        /* close the DRMAA session and exit */
098        err = drmaa_exit(errorstr, DRMAA_ERROR_STRING_BUFFER);
099        if (err != DRMAA_ERRNO_SUCCESS) {
100           printf("Unable to close DRMAA session: %s\n", errorstr);
101        }
102        return err;
103     }
104
105     /* close the DRMAA session and exit */
106     err = drmaa_exit(errorstr, DRMAA_ERROR_STRING_BUFFER);
107     if (err != DRMAA_ERRNO_SUCCESS) {
108        printf("Unable to close DRMAA session: %s\n", errorstr);
109        return err;
110     }
111
112     return 0;
113     }
```

9.6.2 Building a DRMAA Application with Java

When writing a Java DRMAA application it must be taken into account that the Java DRMAA library internally is based on the C DRMAA implementation. The implication is that Java DRMAA is fast, but this native code dependency must be handled properly. The DRMAA application must be run on a submission host with an enabled Altair Grid Engine environment.

Compiling and Running the Java Code DRMAA Example

In order to compile a Java DRMAA application the Java CLASSPATH variable must point to $SGE_ROOT/drmaa/lib/drmaa.jar. Alternatively the -cp or -classpath parameter can be passed to the Java compiler at the time of compilation.

```
> javac -cp $SGE_ROOT/drmaa/lib/drmaa.jar Sample.java
```

To run the application the native code library (libdrmaa.so) must be available in the LD_LIBRARY_PATH environment variable. In this example $SGE_ROOT is expected to be /opt/uge870.

```
> export LD_LIBRARY_PATH=LD_LIBRARY_PATH:/opt/uge870/drmaa/lib/linux
> java -cp $SGE_ROOT/drmaa/lib/drmaa.jar:./ Sample
```

Job Submission, Waiting and Getting the Exit Status of the Job

The following example has the same behaviour as the C example in the section above. First a DRMAA job session is created through a factory method (line 19-22). A new session is opened with the init() call (line 23). After a job template is allocated (line 26) and the remote command parameter (line 29) and the job argument (line 32) is set accordingly, the wait method does not terminate as long the job runs (line 39). Finally the exit status of the job is checked (line 41-47), the job template is freed (line 50) and the session is closed (line 53).

```
000      import java.util.Collections;
001      import org.ggf.drmaa.*;
002
003      public class Sample {
004
005      public static void main(String[] args) {
006
007      Sample sample = new Sample();
008
009       try {
010          sample.example1();
011       } catch (DrmaaException exception) {
012          /* something went wrong */
013          System.out.println("DRMAA Error: " + exception.getMessage());
014       }
015     }
016
017     public void example1() throws DrmaaException {
018     /* get the class, which is needed for creating a session */
019     SessionFactory factory = SessionFactory.getFactory();
020
021      /* create a new session */
022     Session s = factory.getSession();
023     s.init(null);
024
025      /* create a new job template */
026      JobTemplate jobTemplate = s.createJobTemplate();
027
028       /* set "sample.sh" as job script */
029       jobTemplate.setRemoteCommand("/path/to/your/job.sh");
030
031       /* set an additional argument */
032       jobTemplate.setArgs(Collections.singletonList("myarg"));
```

```
033
034        /* submit the job */
035       String jobid = s.runJob(jobTemplate);
036       System.out.println("The job ID is: " + jobid);
037
038       /* wait for the job */
039       JobInfo status = s.wait(jobid, Session.TIMEOUT_WAIT_FOREVER);
040
041       /* check if job exited (and was not aborted) */
042       if (status.hasExited() == true) {
043          System.out.println("The exit code of the job was: "
044                               + status.getExitStatus());
045       } else {
046          System.out.println("The job didn't finish normally.");
047       }
048
049       /* delete the job template */
050       s.deleteJobTemplate(jobTemplate);
051
052       /* close DRMAA session */
053       s.exit();
054    }
055
056  }
```

9.7 Further Information

Java DRMAA related information can be found in the doc directory (HTML format). Further information about DRMAA specific attributes can be found in the DRMAA related man pages:

drmaa_allocate_job_template, drmaa_get_next_attr_value, drmaa_misc, drmaa_synchronize, drmaa_attributes, drmaa_get_next_job_id, drmaa_release_attr_names, drmaa_version, drmaa_control, drmaa_get_num_attr_names, drmaa_release_attr_values, drmaa_wait, drmaa_delete_job_template, drmaa_get_num_attr_values, drmaa_release_job_ids, drmaa_wcoredump, drmaa_exit, drmaa_get_num_job_ids, drmaa_run_bulk_jobs, drmaa_wexitstatus, drmaa_get_attribute, drmaa_get_vector_attribute, drmaa_run_job, drmaa_wifaborted, drmaa_get_attribute_names, drmaa_get_vector_attribute_names, drmaa_session, drmaa_wifexited, drmaa_get_contact,drmaa_init, drmaa_set_attribute, drmaa_wifsignaled, drmaa_get_DRMAA_implementation, drmaa_jobcontrol, drmaa_set_vector_attribute, drmaa_wtermsig, drmaa_get_DRM_system, drmaa_job_ps, drmaa_strerror, jsv_script_interface, drmaa_get_next_attr_name, drmaa_jobtemplate, drmaa_submit

10 Advanced Concepts

Besides the rich set of basic functionality discussed in the previous sections, Altair Grid Engine offers several more sophisticated concepts at time of job submission and during job

execution. This chapter describes such functionality, which becomes important for more advanced users.

10.1 Job Dependencies

In many cases the jobs, which are submitted with Altair Grid Engine are not self-contained. Those jobs are usually arranged in a kind of workflow with more or less complex job dependencies. Such workflows can be mapped to Altair Grid Engine with the submission parameter hold_jid <jobid list>. The <jobid list> contains one or a comma separated list of ids of existing jobs of which the submitted job is waiting for before it can be scheduled. In order get the job IDs, submit the jobs with a name (-N <name>) and use the name as ID. Alternatively the qsub parameter -terse can be used, which transforms the command line result of qsub so that only the job id is returned. This makes it very simple to use within scripts.

10.1.1 Examples

In the following examples, basic workflow control patterns (see www.workflowpatterns.com) are mapped into a Altair Grid Engine job workflow.

Sequence Pattern

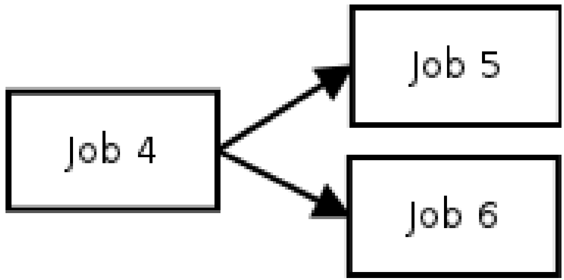
The most simple workflow pattern is the sequence pattern. It is used when a bunch of job must be executed in a pre-defined order. With Altair Grid Engine it is possible to submit all jobs at once but the order is still guaranteed.

```
qsub -b y /bin/sleep 60
Your job 4 ("sleep") has been submitted
qsub -b y -hold_jid 4 /bin/sleep 60
Your job 5 ("sleep") has been submitted
qsub -b y -hold_jid 5 /bin/sleep 60
Your job 6 ("sleep") has been submitted
```

```
> qstat
job-ID  prior   name    user    state submit/start at     queue       slots ja- task-ID
-----------------------------------------------------------------------------------------
  4 0.55500 sleep      daniel     r   03/01/2011 15:16:50 all.q@host1         1
  5 0.00000 sleep      daniel    hqw  03/01/2011 15:17:52                     1
  6 0.00000 sleep      daniel    hqw  03/01/2011 15:17:57                     1
```

Parallel Split/Fork Pattern

The fork pattern is used when a job sequence involves tasks that are executed in parallel. In this case two or more jobs depend on just one job, meaning they are scheduled after the job is complete. In Altair Grid Engine , this is mapped through setting the hold job ID value of multiple jobs to the same job.



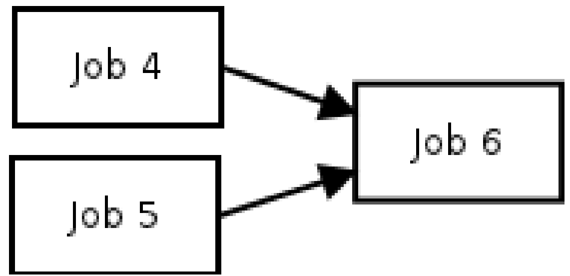

```
qsub -terse -b y /bin/sleep 60
4
qsub -b y -hold_jid 4 /bin/sleep 60
Your job 5 ("sleep") has been submitted
qsub -b y -hold_jid 4 /bin/sleep 60
Your job 6 ("sleep") has been submitted

> qstat
job-ID  prior   name  user  state  submit/start at     queue       slots ja-task-ID
------------------------------------------------------------------------------------
    4 0.55500 sleep  daniel   r    03/01/2011 16:00:50 all.q@host1        1
    5 0.00000 sleep  daniel  hqw   03/01/2011 16:00:58                    1
    6 0.00000 sleep  daniel  hqw   03/01/2011 16:01:00                    1

> qstat
job-ID  prior   name  user  state submit/start at     queue       slots ja-task-ID
------------------------------------------------------------------------------------
    5 0.00000 sleep  daniel   r   03/01/2011 16:00:58                     1
    6 0.00000 sleep  daniel   r   03/01/2011 16:01:00                     1
```

In this example job 5 and 6 depending from job 4.

After job 4 finishes both jobs are scheduled within the same time.

Synchronization Pattern

With the synchronization pattern, a job starts (is scheduled) when all dependencies are fulfilled, i.e. that all of the waiting jobs have completed. It is usually used after parallel sections induced by the parallel split/fork pattern or when a job is one which finalizes the work of multiple jobs (post processing).




```
qsub -b y /bin/sleep 60
Your job 4 ("sleep") has been submitted
qsub -b y /bin/sleep 120
Your job 5 ("sleep") has been submitted
qsub -b y -hold_jid 4,5 /bin/sleep 60
Your job 6 ("sleep") has been submitted

> qstat
job-ID  prior   name    user   state submit/start at     queue      slots ja-task-ID
-------------------------------------------------------------------------------------
  4 0.55500 sleep      daniel   r    03/01/2011 16:24:50 all.q@host1         1
  5 0.00000 sleep      daniel   r    03/01/2011 16:24:54 all.q@host2         1
  6 0.00000 sleep      daniel   hqw  03/01/2011 16:24:57                     1

> qstat
job-ID  prior name   user   state submit/start at      queue       slots ja-task-ID
------------------------------------------------------------------------------------
  5 0.00000 sleep   daniel    r    03/01/2011 16:24:54 all.q@host2           1
  6 0.00000 sleep   daniel   hqw   03/01/2011 16:24:57                       1
```

In this example the job 6 depends on two previously submitted jobs. The hold state of the job is removed not before job 4 and job 5 ended.

10.2 Using Environment Variables

During job execution a number of environment variables are set from Altair Grid Engine and are available for the executing script/binary. These variables contain information about Altair Grid Engine specific settings, job submission related information and other details. Additionally the user can specify at time of submission using the -v and -V parameter self-defined environment variables. While -v expects a list of variable=value pairs, which are passed-through from job submission to the jobs environment, the -V parameter transfers all environment variables from the job submission context into the jobs execution context.

```
qrsh -v answer=42 myscript.csh
```

In myscript.csh *$answer* has the value 42.

```
setenv answer 42
qrsh -V myscript.csh
```

In myscript.csh *$answer* has the value 42.

In the following tables all Altair Grid Engine environment variables available during job execution are listed:

Table 34: Standard Job Environment Variables

Variable Name	Semantic
SGE_ARCH	The architecture of the host on which the job is running.
SGE_BINARY_PATH	The absolute path to the Altair Grid Engine binaries.
SGE_JOB_SPOOL_DIR	The directory where the Altair Grid Engine shepherd stores information about the job.
SGE_JSV_TIMEOUT	Timeout value (in seconds), when the client JSV will be restarted.
SGE_STDERR_PATH	The absolute path to the standard error file, in which Altair Grid Engine writes errors about job execution.
SGE_STDOUT_PATH	The absolute path to the standard output file, in which Altair Grid Engine writes the output of the job.
SGE_STDIN_PATH	The absolute path to file, the job uses as standard input.
ENVIRONMENT	Altair Grid Engine fills in BATCH to identify it as an Altair Grid Engine job submitted with qsub.
HOME	Path to the home directory of the user.
HOSTNAME	Name of the host on which the job is running.
JOB_ID	ID of the Altair Grid Engine job.
JOB_NAME	Name of the Altair Grid Engine job.
JOB_SCRIPT	Name of the script, which is currently executed.
LOGNAME	Login name of the user running the job on the execution host.
PATH	The default search path of the job.
QUEUE	The name of the queue in which the job is running.
REQUEST	The name of the job specified with the -N option.
RESTARTED	Indicates if the job was restarted (1) or if it is the first run (0).
SHELL	The login shell of the user running the job on the execution host.
TMPDIR	The absolute path to the temporary directory on the execution host.
TMP	The absolute path to the temporary directory on the execution host.
TZ	The timezone set from the execution daemon.
USER	The login name of the user running the job.

Table 35: Job Submission Related Job Environment Variables

Variable Name	Semantic
SGE_O_HOME	The home directory on the submission host.
SGE_O_HOST	The name of the host, on which the job is submitted.
SGE_O_LOGNAME	The login name of the job submitter.
SGE_O_MAIL	The mail directory of the job submitter.

Variable Name	Semantic
SGE_O_PATH	The search path variable of the job submitter.
SGE_O_SHELL	The shell of the job submitter.
SGE_O_TZ	The time zone of the job submitter.
SGE_O_WORKDIR	The working directory path of the job submitter.

Table 36: Parallel Jobs Related Job Environment Variables

NHOSTS	The number of hosts on which this parallel job is executed.
NQUEUES	The number of queues on which this parallel job is executed.
NSLOTS	The number of slots this parallel job uses (1 for serial jobs).
PE	Only available for parallel jobs: The name of the parallel environment in which the job runs.
PE_HOSTFILE	Only available for parallel jobs: The absolute path to the pe_hostfile.

Table 37: Checkpointing Jobs Related Job Environment Variables

SGE_CKPT_ENV	Checkpointing jobs only: Selected checkpointing environment.
SGE_CKPT_DIR	Checkpointing jobs only: Path of the checkpointing interface.

Table 38: Array Jobs Related Job Environment Variables

SGE_TASK_ID	The task number of the array job task the job represents. If the job is not an array task, the variable contains undefined.
SGE_TASK_FIRST	The task number of the first array job task. If the job is not an array task, the variable contains undefined.
SGE_TASK_LAST	The task number of the last array job task. If the job is not an array task, the variable contains undefined.
SGE_TASK_STEPSIZE	Contains the step size of the array job. If the job is not an array task, the variable contains undefined.

10.3 Using the Job Context

Sometimes it is necessary that a job makes its internal state visible to qstat. This can be done with the job execution context. Job context variables can be initially set on job submission time with the -ac name=value parameter and altered/added and deleted during run-time with qalter -ac or -dc switch. In the following example a job script makes the internal job state visible to the qstat client. The context_example.sh job script looks like the following:

```
00   #!/bin/sh
01
02   sleep 15
```

```
03
04   $SGE_BINARY_PATH/qalter -ac STATE=staging $JOB_ID
05
06   sleep 15
07
08   $SGE_BINARY_PATH/qalter -ac STATE=running $JOB_ID
09
10   sleep 15
11
12   $SGE_BINARY_PATH/qalter -ac STATE=finalizing $JOB_ID
```

Now the job with the context STATE=submitted is submitted and the context is filtered with the grep command every 15 seconds.

```
> qsub -ac STATE=submitted context_example.sh
Your job 4 ("context_example.sh") has been submitted
> qstat -j 4 | grep context
context:                    STATE=submitted
> sleep 15
> qstat -j 4 | grep context
context:                    STATE=staging
> sleep 15
> qstat -j 4 | grep context
context:                    STATE=running
> sleep 15
> qstat -j 4 | grep context
context:                    STATE=finalizing
```

10.4 Transferring Data

A common way to transfer input and output data to and from the user application is to use a distributed or network file system like NFS. While this is easy to handle for the user applications, the performance can be a bottleneck, especially when the data is accessed multiple times sequentially. Hence Altair Grid Engine provides interfaces and environment variables for delegated file staging in order to support the user with hook points for accessing and transferring data in different ways. In the following section these approaches for transferring user data as well as their advantages and drawbacks are discussed.

10.4.1 Transferring Data within the Job Script

While the job script is transferred from the submission host to the execution host, the data the job is working on remains unknown and therefore unreflected by the qsub command. If the necessary input and output files are only available through a slow network file system on the execution host, they can be staged in and out from the job itself to the local host. In order to do so, Altair Grid Engine creates a local temporary directory for each job and deletes it automatically after the job ends. The absolute path to this local directory is as $TMPDIR environment variable available during job run-time. In the following example an

I/O intensive job copies the input data set from the NFS exported home directory of the user to the local directory and the results back to the home directory.

```
#!/bin/sh
...
# copy the data from the exported home directory to the temporary directory
cp ~/files/largedataset.csv $TMPDIR/largedataset.csv
# do data processing
...
# copy results back to user home directory
cp $TMPDIR/results ~/results
```

10.4.2 Using Delegated File Staging in DRMAA Applications

The Altair Grid Engine DRMAA implementation comes with built-in support for file staging. The administrator must configure appropriate prolog and epilog scripts, which are executed before the DRMAA jobs starts and after the DRMAA job ends. Theses scripts can be configured in the global configuration (qconf -mconf), in the host configuration (qconf -mconf hostname), and in the queue configuration (qconf -mq queuename). The script that is executed depends on the scripts which are configured. The host configuration overrides the global configuration and the queue configuration dominates the host configuration.

In order to make the job and epilog script job obvious, a set of variables is defined by Altair Grid Engine. These variables can be used in the configuration line, where the path to the pro- and epilog is defined.

: Delegated File Staging Variables

Example: Copy the DRMAA Job Output File

In the following example an epilog script is parameterized in a way that the DRMAA job output file is copied after the job ends to an user defined host and directory.

```
qconf -mconf
...
epilog /path/to/epilog.sh $fs_stdout_file_staging $fs_stdout_host $fs_stdout_path
$fs_stdout_tmp_path
...
```

The epilog.sh script looks like the following:

```
000     #!/bin/sh
001
002     doFileStaging=$1
003     outputHost=$2
004     outputHostPath=$3
005     tmpJobPath=$4
006
007     if [ "x$doFileStaging" = "x1" ]; then
```

```
008
009      # transfer file from execution host to host specified in DRMAA file
010      echo "Copy file $tmpJobPath to host $outputHost to $outputHostPath"
011      scp $tmpJobPath $outputHost:$outputHostPath
012
013      fi
```

Finally the DRMAA delegated file staging must be turned on:

```
qconf -mconf
 ...
delegated_file_staging true
```

After this is configured by the Altair Grid Engine administrator everything is the prepared from the Altair Grid Engine side. The DRMAA application now has to determine where to copy the information of the output file in the job template. The following code example shows how to accomplish this with Java DRMAA.

```
/* enable transfer output file to host "yourhostname" in file "/tmp/JOBOUTPUT" */
jobTemplate.setOutputPath("yourhostname:/tmp/JOBOUTPUT");
/* disable transfer input file, enable transfer output file, disable transfer error file *
FileTransferMode mode = new FileTransferMode(false, true, false);
jobTemplate.setTransferFiles(mode);
```

Go back to the Altair Grid Engine Documentation main page.

10.5 Manual, Semi-Automatic and Automatic Preemption

Altair Grid Engine clusters can cope with different types of competing workloads. The configuration of the Altair Grid Engine scheduler determines how different workloads will be scheduled in the system. Policies can be combined to achive almost any type of scheduling.

In previous versions of Altair Grid Engine enforcing policies was sometimes difficult especially when high priority jobs required resources of lower priority jobs that already consumed resources like slots, memory or licenses. In such cases slot-wise suspend on subordinate queues was used to release resources in use by other jobs or reservation and advance reservation functionality could be used to reserve resources for high priority jobs while they are pending in the system.

Altair Grid Engine 8.3 (and above) now provides the functionality to release resources when required resources are already in use. This can be done through preemption. This section describes preemptive scheduling as an addition to the Altair Grid Engine job handling and scheduling that now makes it possible for high priority work to force the release of resources in order to run in the cluster.

10.5.1 Preemption Terms

Following paragraphs describe a couple of terms that are used throughout this section.

Jobs which have high priority based on the configured policies can get the role of a *preemption consumer* that can cause a *preemption action* to be performed for one or more running jobs that have the role of a *preemption provider*. In general all those running jobs are considered as *preemption provider* where the priority is lower than that of the *preemption consumer*.

There are different preemption actions available in Altair Grid Engine. What all of them have in common is that they will make all or a subset of the bound resources of a *preemption provider* available so that they can be used by one or more *preemption consumer*. Different *preemption actions* differ in the way how bound resources are freed and how the Altair Grid Engine system will make the bound resources available.

Preemption actions can be executed by Altair Grid Engine due to three different *preemption triggers*. A *preemption trigger* will define the time and has an influence on the chosen *preemption action* that is performed. In general *preemption trigger* can be manual, semi-automatic or automatic.

A *preemption consumer* that consumes resources provided through triggering a *preemption action* has the role of a *preemptor* whereas those jobs that get forced to free resources are considered as *preemptee*.

In Altair Grid Engine 8.3 manual preemption is implemented. semi-automatic or automatic trigger will follow with upcoming releases.

10.5.2 Preemption Trigger and Actions

Altair Grid Engine 8.3 provides six different preemption actions to preempt a job. With manual preemption the user/manager has to choose which of the available preemptive actions should be used to trigger preemption of a job. With semi-automatic and automatic preemption mechanisms (available with future versions of Altair Grid Engine) either the system configuration or the Altair Grid Engine scheduler decides automatically which preemption action will be taken to release resources.

The six preemptive actions differ in the way the resources will be available for other jobs after the preemptee is preempted. Some of those actions have restrictions on which job types they can be applied to as well as who is allowed to trigger them. The actions differ in the way how they treat the processes that are executed on behalf of a job that gets preempted.

Within Altair Grid Engine all preemption actions are represented by single capital letter (**T**, **R**, **C**, **P**, **N** or **S**) that is either passed to a command, specified in a configuration object or is shown in command output displaying the internal state of a job.

Some of the preemption actions trigger the **suspend_method** that might be defined in the queue where the preemptee is executed. To be able to distinguish different preemption actions within the **suspend_method** an optional argument named

$action * *maybeusedaspseudoargumentwhenthemethodisdefined.The * *$**action** argument will be expanded to the corresponding letter that represents the preemptive action during runtime.

Terminate Action: The preemptee will be terminated. As soon as all underlying processes are terminated all resources that were bound by that preemptee will be reported as free. The **T**-action can be applied to any job. Users can apply it only to their own jobs.

Checkpoint Action: The preemptee will be checkpointed. As soon as a checkpoint is written and all underlying processes are terminated all bound resources will be reported as available and the job will be rescheduled. This preemption action can only be applied to checkpointable jobs where a checkpointing environment was specified during submission of the job.

Rerun Action: The preempted job will be rescheduled. As soon as all underlying processes are terminated all bound resources will be reported as available. Managers can enforce the rerun of jobs even if those jobs are not tagged as rerun-able on the job or queue level.

Preemption Action: The preemptee will be preempted. Preempted means that the configured queue-suspend method (**$action** set to **P**) will be executed that might trigger additional operations to notify the processes about the upcoming preemption so that those processes can release bound resources by itself. After that the processes are suspended and all consumable resources, where the attribute available-after-preemption (**aapre**) is set to true, are reported as free. Not-available-after-preemption resources are still reported to be bound by the preempted job. The preemption action can be applied to all preemption providers whereas users can only preempt their own jobs.

e**N**hanced Suspend Action: Similar to the preempt action the queue **suspend_method** (**$action** set to **N**) will be triggered before the preemptee gets suspended. Only non-memory-based consumables (including License Orchestrator managed license resources) are reported as free when the processes are suspended. Memory-based consumables that are available-after-preemption and also not-available-after-preemption consumables will still be reported as bound by the enhanced suspended job. This preemption action can be applied to all preemption providers. Users can only preempt their own jobs.

Suspend Action: Similar to the preempt action the triggered method will be the **suspend_method** (**$action** set to **S**) before the preemptee gets suspended. Only consumed slots (and License Orchestrator-managed license resources) will be available after suspension. All other resources, independent if they are tagged as available-after-preemption or not-available-after-preemption in the complex configuration, will be reported as still in use. This preemption action can be applied to all preemption providers. Users can only preempt their own jobs.

The obvious question regarding preemption is; which of the six preemptive actions should be chosen to manually preempt a job? If a job is checkpointable then it should be the **C**-action. Here all consumed resources of the preemptee will be available for higher priority jobs. The preemptee can continue its work when it is restarted from the last written checkpoint.

The **T**-action and the **R**-actions release the full set of resources but they should be seen as the last resort when no other less disruptive preemptive actions can be applied. The reason for this is that the computational work of the preemptee up to the point in time where the preemptee is rescheduled or terminated is typically lost which is a waste of cluster resources.

From the Altair Grid Engine perspective the **P**-action makes all bound resources (slots + memory + other consumable resources where **aapree** of the complex is set to true) available for higher priority jobs. But this operation is only correct if the machine has enough swap space configured so that the underlying operating system is able to move consumed physical memory pages of the suspended processes into that swap space and when the application either releases consumed resources (like software licenses, special devices, ...) automatically or when a **suspend_method** can be configured to trigger the release of those resources. The **N**-action can be used for jobs that run on hosts without or with less configured swap space. The **N** action will release only non-memory-based consumables (slots + other consumable resources where **aapree** of the complex is set to true).

If jobs either do not use other resources (like software licenses, special devices, ...) and when in use memory on the node does not need to be released, then the **S**-action can be chosen. It is the simplest preemption action that provides slots (and License Orchestrator licenses) only after preemption. Please note that the **S**-action and **S**-state of jobs is different from the s-state of a job (triggered via **qmod -s** command). **A regularly suspended job does not release slots of that job**. Those slots will remain in use by the job that was suspended.

The **P** and **N**-action will make consumable resources of preemptees available for higher priority jobs. This will be done automatically for all preconfigured consumable resources in a cluster. For those complexes the available-after-preemption-attribute (**aapre**) is set to **YES**. Managers of a cluster can change this for predefined complexes. They also have to decide if a site defined resource is available after preemption. For resources that should be ignored by the preemptive scheduling functionality the **aapre**-attribute can be set to **NO**.

Please note that the resource set for each explained preemptive action defines the maximum + set of resources that might get available through that preemption action. Additional scheduling parameters (like **prioritize_preemptees** or **preemptees_keep_resources** that are further explained below) might reduce the resource set that get available through preemption to a subset (only those resources that are demanded by a specified **preemption_consumer**) of the maximum set.

10.5.3 Manual Preemption

Manual preemption can be triggered with the qmod command in combination with the *-p* command line switch. The *-p* expects one job ID of a *preemption_consumer* followed by one or multiple job IDs or job names of *preemption_provider*. The last argument contains an optional character representing one of the six *preemptive_actions*. When the last argument is omitted **P**-action will be used as default.

Syntax:

```
qmod [-f] -p <preemption_consumer>
          <preemption_provider> [<preemption_provider> ...]
          [<preemption_action>]

<preemption_consumer> := <job_ID> .
<preemption_provider> := <job_ID> | <job_name> .
<preemption_action>   := "P" | "N" | "S" | "C" | "R" | "T" .
```

The manual preemption request will only be accepted if it is valid. Manual preemption request will be rejected when:

- Resource reservation is disabled in the cluster.
- *preemption_consumer* has no reservation request.
- At least one specified *preemption_provider* is not running.
- **C**-action is requested but there is at least one *preemption_provider* that is not checkpointable.
- **R**-action is requested but there is at least one *preemption_provider* that is neither tagged as rerunnable nor the queue where the job is running is a rerunnable queue. (Manager can enforce the **R**-action in combination with the *-f* command line argument).

Manual preemption requests are not immediately executed after they have been accepted by the system. The Altair Grid Engine scheduler is responsible for triggering manual preemption during the next scheduling run. Preemption will only be triggered if the resources are not available to start the preemption consumer within a configurable time frame (see **preemption_distance** below). If enough resources are available or when the scheduler sees that they will be available in near future then the manual preemption request will be ignored.

Please note that resources available through preemption are only reserved for the specified *preemption_consumer* as long as there are no other jobs of higher priority that demand those resources. If there are jobs of higher priority then those jobs will get the resources and the specified *preemption_consumer* might stay in pending state until either the higher priority jobs leaves the system or another manual preemption request is triggered.

Preemptees will automatically trigger a reservation of all resources lost due to preemption. This means that Preemptees can be reactivated as soon as they are eligible due to priority and as soon as the missing resources are available. There is no dependency between a preemptor and the preemptees. All or a subset of preemptees might get restarted even if the preemptor is still running if requested resources are added to the cluster or become available due to other jobs completing.

Preemtees will have the jobs state **P**, **N** or **S** (shown in the *qstat* output or *qmon* dialogs) depending of the corresponding preemption action that was triggered. Those jobs, as well as preemptees that are rescheduled due to the **R**-action, will appear as pending jobs even if they still hold some resources. Please note that regularly suspended jobs (in s-state due to qmod -s) still consume all resources and therefore block the queue slots for other jobs. qstat -j command can be used to see which resources are still bound by preemptees.

10.5.4 Preemption Configuration

The following scheduling configuration parameters are available to influence the preemptive scheduling as well as the preemption behaviour of the Altair Grid Engine cluster.

max_preemptees: The maximum number of preemptees in the cluster. Preempted jobs may hold some resources such as memory and if the **preemptees_keep_resources** parameter is configured might keep most of their resources while in a preempted state. A high

number of preemptees can significantly impact cluster operation and throughput. Limiting the number of preemptees will limit the amount of held but unused resources.

prioritize_preemptees: By setting this parameter to *true* or *1* preemptees get a reservation before the regular scheduling is done. This can be used to ensure that preemptees get restarted again at the earliest possibly opportunity when the preemptor finishes, unless resources required by the preemptee are still held by jobs which were backfilled. *prioritize_preemptees* in combination with disabling backfilling provides a guarantee that preemptees get restarted when the preemptor finishes, at the expense of lower cluster utilization.

preemptees_keep_resources: When a job gets preempted the freed resources will only be consumed by the preemptor. This prevents resources of a preemptee from being consumed by other jobs. **prioritize_preemptees** and **preemptees_keep_resources** in combination provide a guarantee that preemptees get restarted as soon as the preemptor finishes, at the expense of a waste of resources and bad cluster utilization. Exception: Licenses managed through License Orchestrator and a license manager cannot be held by a preemptee. As the **preemptee processes** are suspended the license manager might assume the license is free which will lead to the license be consumed by a different job. When the preemptee processes get unsuspended a license query will fail if the license is held.

preemption_distance: A preemption will only be triggered if the resource reservation that could be created for a job is farther in the future than the given time interval (hh:mm:ss default 00:15:00). Reservation can be disabled by setting the value to 00:00:00. No Reservation will be created if job preemption is forced by a manager manually using *qmod -f -p ...* .

10.5.5 Preemption in Combination with License Orchestrator

License complexes that are reported by License Orchestrator are automatically defined as available-after-preemption (**aapre** is set to **YES**). This means that when a Altair Grid Engine job that consumes a License Orchestrator license resource gets preempted, it triggers an automatic preemption of the corresponding License Orchestrator license request. The license will be freed and is then available for other jobs.

Manual preemption triggered in one Altair Grid Engine cluster does not provide a guarantee that the specified preemption consumer (or even a different job within the same Altair Grid Engine cluster) will get the released resources. The decision which cluster will get the released resource depends completely on the setup of the License Orchestrator cluster. Consequently it might happen that a license resource that gets available through preemption in one cluster will be given to a job in a different cluster if the final priority of the job/cluster is higher than that of the specified preemption consumer.

10.5.6 Common Use Cases

License consumable (without License Orchestrator) Scenario: There is a license-consumable defined that has a maximum capacity and multiple jobs compete for the license-consumable by requesting one or multiple of those licenses.

Complex definition:

```
$ qconf -sc
...
license  lic  INT  <=  YES  YES  0  0  YES 0.000000 YES       NO
...
```

The nineth attribute **YES** defines the value of **aapre**. This means that the license resource will be available after preemption.

License capacity is defined on global level:

```
$ qconf -se global
...
complex_values   license=2
```

When two jobs are submitted into the cluster both licenses can be consumed by the jobs.

```
$ qsub -l lic=1 -b y -l h_rt=1:00:00 sleep 3600
$ qsub -l lic=1 -b y -l h_rt=1:00:00 sleep 3600
...

$ qstat -F lic
...
all.q@rgbtest                  BIPC  0/1/60   lx-amd64
       gc:license=0
3000000005 0.55476 sleep      user         r
---------------------------------------------------------------------------------
all.q@waikiki                  BIPC  0/1/10         0.00     lx-amd64
       gc:license=0
3000000004 0.55476 sleep      user         r     04/02/2015 12:32:54     1
```

Submission of a higher priority job requesting 2 licenses and resource reservation:

```
$ qsub -p 100 -R y -l lic=2 -b y -l h_rt=1:00:00 sleep 3600
```

The high priority job stays pending, it will get a reservation, but only after both lower priority jobs are expected to finish:

```
$ qstat -j 3000000006
...
reservation       1:    from 04/02/2015 13:33:54 to 04/02/2015 14:34:54
                        all.q@hookipa: 1
```

We want the high priority job to get started immediately, therefore we trigger a manual preemption of the two lower priority jobs:

```
$ qmod -p 3000000006 3000000004 3000000005 P
Accepted preemption request for preemptor candidate 3000000006
```

The lower priority jobs get preempted, the high priority job can start:

```
$ qstat
job-ID     prior   name  user state submit/start at     queue     jclass slots ja-task-ID
------------------------------------------------------------------------------------------
3000000006 0.60361 sleep joga r 04/02/2015 12:37:50 all.q@waikiki        1
3000000004 0.55476 sleep joga P 04/02/2015 12:32:54                      1
3000000005 0.55476 sleep joga P 04/02/2015 12:32:54                      1
```

Resources which have been preempted are shown in qstat -j. In order for the preemptees to be able to resume work as soon as possible, preempted jobs get a resource reservation for the resources they released, e.g.

```
$ qstat -j 3000000004
...
preempted   1: license, slots
usage       1: wallclock=00:04:45, cpu=00:00:00, mem=0.00015 GBs, io=0.00009,
               vmem=19.414M, maxvmem=19.414M
reservation 1: from 04/02/2015 13:38:50 to 05/09/2151 19:07:05
               all.q@waikiki: 1
```

11 Submitting Jobs from or to Windows hosts

Registering User passwords

In order to execute a job on a Windows execution host, Altair Grid Engine has to log on as the job user on this host. For this, Altair Grid Engine needs the user's password. Use the `sgepasswd` command on a UNIX Altair Grid Engine submit or administrative host to register the user password. The `sgepasswd` command encrypts the password and stores it in the `$SGE_ROOT/$SGE_CELL/common/sgepasswd` file.

Network shares on the Windows execution host

If a mounted network drive (`net use x: \\server\share`) is used in the path to the job binary or in some argument to the job, this often does not work. Even if the job user has persistent mounts on the Windows execution host or if the job user is logged on to the Windows execution host and has created some mounts manually, these are not available for the job. The job runs in a separate session, manually created mounts do not exist there, and persistent mounts show inconsistent behaviour, so they should be avoided, too. Instead, UNC paths should be used.

E.g.

```
> net use x: \\server\share
> qsub -b y x:\path\job.exe
```

will not work, while

```
> qsub -b y \\server\share\path\job.exe
```

should work.

Submitting Jobs from or to Windows hosts

If a job is submitted from or to a Windows host, it has to be taken care of some of the differences between UNIX and Windows. It has to be distinguished between three different cases:

1. Job submission from a Windows submit host to a Windows execution host
2. Job submission from a Windows submit host to a UNIX execution host
3. Job submission from an UNIX submit host to a Windows execution host

So generally spoken, if the cluster consists of a mixture of Windows and UNIX hosts, for all jobs the destination architecture should be specified, either directly or indirectly.

11.1 Job submission from a Windows submit host to a Windows execution host

This is an example job that contains all elements that differ from a normal UNIX-to-UNIX job submission:

```
> qsub -o /tmp/$JOB_ID.out -b y cmd.exe /c %SGE_ROOT%\examples\jobs\sleeper.bat
```

It contains:

- Paths in arguments to the submit client have to be in UNIX format: `-o /tmp/$JOB_ID.out`
- Prevent job script transfer: `-b y`
- The job itself, which is the interpreter for the script: `cmd.exe`
- Paths in arguments to the job have are not mapped by Altair Grid Engine: `%SGE_ROOT%\examples\jobs\sleeper.bat`

Paths in arguments have to be in UNIX format

The Altair Grid Engine submit clients cannot handle paths in Windows/DOS or UNC format. The parser of these clients will reject all paths in arguments to the client itself that are not in UNIX format. Thus, it is necessary to specify these paths in UNIX format.

If these paths are to be evaluated on the Windows execution host, the path mapping file must contain the corresponding entry, so the execution daemon can translate the UNIX path to the Windows path.

In this example, the variable `$JOB_ID` is used in the path. This works, because this variable is set on the Windows execution host in the execution daemon. The execution daemon resolves these variables in UNIX format properly, no matter if on Windows or UNIX.

Prevent job script transfer

This is necessary because the job - `cmd.exe` - is a binary that is already located on the Windows execution host.

The job itself - the interpreter for the script

On Windows, there is no hashbang magic, so the submitter must tell the execution daemon what interpreter there is to be started in order to execute the "sleeper.bat" script. The "/c `%SGE_ROOT%\examples\jobs\sleeper.bat`" are just arguments to the interpreter. The interpreter already exists on the execution host, so Altair Grid Engine does not have to transfer it from the submit host to the execution host. This is denoted by the "-b y" option.

Paths in the job arguments are not touched by Altair Grid Engine

Altair Grid Engine has no idea what the arguments to the job mean or where they are evaluated, so Altair Grid Engine cannot map paths in the arguments to the job. They have to be specified in the format that can be used by the job itself.

Examples:

- Submit a binary job:

```
> qsub -o /home/jdoe/outfile.txt -j y -N myjob -b y notepad.exe
```

- Submit an interactive job:

```
> qrsh hostname
```

11.1.1 Running Jobs in the foreground

Some Windows applications do not start in the background, i.e. if they are not started in a Windows session that has a visible desktop. This desktop may be visible on the screen that is physically attached to the host, or in any Remote Desktop session or similar. Furthermore, some Windows applications open a MessageBox in case of errors, even if they are designed to run in background. Also Windows itself might show a MessageBox if e.g. a DLL is missing to run the application.

In order to allow these jobs to run or to see the error MessageBoxes, Altair Grid Engine allows to start jobs in the foreground, even on a foreign desktop. A job can request to be started in the foreground by requesting the `display_win_gui` attribute (short: `dwg`), which is of type `BOOL`.

Example:

```
> qsub -l display_win_gui=true -b y notepad.exe
```

should start the Windows notepad on a Windows execution host, allowing it to show it's window on the currently visible desktop.

If no desktop is visible at all, this job will fail!

Running foreground jobs in the background

It is also possible to force a job to run in the background even if it requests the `display_win_gui` attribute. This can be used in job dependencies - if two jobs have to run on the same host and the second job has to run in the foreground, the first still can run in the background, but the Scheduler of Altair Grid Engine needs the information that it has to select a Windows execution host that provides `display_win_gui=true` already for the first job.

This feature is enabled by setting the `SGE_BACKGND_MODE` in the job environment to `1`, e.g.:

```
> qsub -l display_win_gui=true -v SGE_BACKGND_MODE=1 -b y notepad.exe
```

Because this doesn't work for certain applications which put themselves in the foreground, there is an option to redirect the GUI of such jobs to an invisible Window station and desktop. This feature is enabled by setting the `SGE_DWG_DESKTOP` in the job environment accordingly. These settings are supported: * `service` to use a desktop the services Window station in logon session 0 to create a new desktop for the job's GUI there. The `SGE_BACKGND_MODE` variable is ignored in this case. * `none`, any unsupported value or omit the `SGE_DWG_DESKTOP` environment variable to use the old behaviour, i.e. let Altair Grid Engine search for a desktop to display the GUI on. `SGE_BACKGND_MODE` is obeyed like described above.

E.g.:

```
> qsub -l display_win_gui=true -v SGE_DWG_DESKTOP=service -b y notepad.exe
```

starts the job in session 0 (the services session), creates a new Window station and desktop for the GUI of the job and associates the job with this desktop. The desktop is invisible but still the job application can use all functionality a Window station and desktop provide.

11.2 Job submission from an UNIX submit host to a Windows execution host

In general, the same rules apply as for job submission from a Windows submit host to a Windows execution host. The only thing to take care of is the shell on the submit host, if a path in Windows format is specified on the command line, the shell will consume the backslashes, so the backslashes have to be doubled. Furthermore, variables that are to be resolved on the submit host have to be specified in UNIX format (starting with a `$` sign) - but take care to which format variables that contain a path do resolve! The path in the job argument of the test job:

```
# qsub -b y cmd.exe /c $SGE_ROOT\\examples\\jobs\\sleeper.bat
```

would resolve to e.g. `/opt/uge\examples\jobs\sleeper.bat`, which doesn't exist on the Windows execution host, so the proper way to specify this path is to manually map the path and specify it as an absolute path:

```
# qsub -b y cmd.exe /c \
  \\\\server\\opt\\uge\\examples\\jobs\\sleeper.bat
```

which resolves to \\server\opt\uge\examples\jobs\sleeper.bat which exists on the Windows execution host.

If a Altair Grid Engine variable is used as argument to an Altair Grid Engine option, i.e. if it has to be resolved on the execution host, not on the submit host, the $ sign has to be escaped, too:

```
# qsub -o /tmp/\$JOB_ID.out -b y cmd.exe /c \
  \\\\server\\opt\\uge\\examples\\jobs\\sleeper.bat
```

11.3 Job submission from a Windows submit host to an UNIX execution host

These jobs can be submitted like jobs from UNIX to UNIX, only paths in arguments to the job have to be mapped manually and again, paths in variables are resolved on the submit host, likely to the wrong format:

```
> qsub %SGE_ROOT%/examples/jobs/sleeper.bat
```

would resolve to \\server\opt\uge/examples/jobs/sleeper.bat which doesn't exist on the submit host, so again manual mapping is needed:

```
> qsub /opt/uge/examples/jobs/sleeper.bat
```

resolves to /opt/uge/examples/jobs/sleeper.bat, which should perfectly work on the UNIX execution host.