Nitro 2.0

User Guide

March 2016

Adaptive
COMPUTING

Scan to open online help

Welcome

Welcome to the User Guide for Nitro 2.0.

The following chapters are provided to assist in understanding, and using Nitro.

- Nitro Overview on page 3 - Provides basic information on Nitro, including theory of operation.

- Using Nitro on page 9 - Contains procedures and reference information on using Nitro.

- References on page 23 - Provides additional conceptual information about Nitro, including a glossary of key terms used throughout this guide.

- Troubleshooting on page 35 - Identifies common sources of reference for troubleshooting and provides troubleshooting information for task errors.

Chapter 1 Nitro Overview

Nitro is the Adaptive Computing High Throughput Computing (HTC) product designed to integrate with either High Performance Computing (HPC), such as Moab Workload Manager, or datacenter schedulers to schedule and run workloads consisting of large quantities (tens of thousands to millions) of small jobs (seconds to minutes to complete) without affecting the throughput of the HPC or datacenter scheduler.

In this chapter:

Introduction to Nitro

This Nitro User Guide documentation explains how to define tasks for a task file, the kinds of information you can give to Nitro via your job script and/or job submission to customize your Nitro job's execution, and the information Nitro will give you as it executes your tasks and after it has finished.

Nitro lets you execute many workloads quickly using a single job.

This topic introduces you to Nitro and provides some basic understanding about how to use Nitro.

In this topic

What is Nitro?

Nitro is a "high-throughput task scheduler" product that quickly executes user jobs or "tasks" that fit the criterion of being able to fit and execute on a single host.

A job or task can be a

- single-core serial application
- multi-threaded application
- small parallel application
- regression test
- embarrassingly parallel" application such as a Monte Carlo simulation.

While Nitro can be used for workloads that execute in short (sub-second) or long (hours) periods of time, the shorter a workload executes, the better performance gain Nitro will give you over a normal job scheduler; meaning the shorter the workload, the higher the performance gain.

Notwithstanding the advantages for small and short jobs, some sites where users may submit many (1000s) small jobs use Nitro to improve their normal job scheduler's performance, which degrades when the site has a large queue containing tens or hundreds of thousands, or even millions, of jobs. These sites have these users convert their many small jobs into a single Nitro job, which can greatly reduce the job queue size and speeds the response time for the users since the normal job scheduler can schedule a single job that then executes thousands or millions of small jobs as "tasks".

How Nitro Works

To execute many workloads quickly, you give Nitro a text-based "task" file containing commands that execute application programs you want to run. The task file has no limit on the number of application program commands, which means the task file may contain 10, 10 thousand, or 10 million tasks; it does not matter to Nitro.

If you have many small jobs, you convert them into Nitro tasks by extracting the jobs' commands to execute application programs and then putting the extracted commands into a single Nitro task file.

To use Nitro, you must do these four things:

1. Create a task file.

2. Create a job script.

3. Submit your job script to a job scheduler and give it the location of your task file.

4. Examine the Nitro job output to see the results.

About a Nitro Job

The following image identifies and illustrates the different items involved with a Nitro job, its submission to a job scheduler, its execution on the job's hosts, and the results it generates and returns to you.
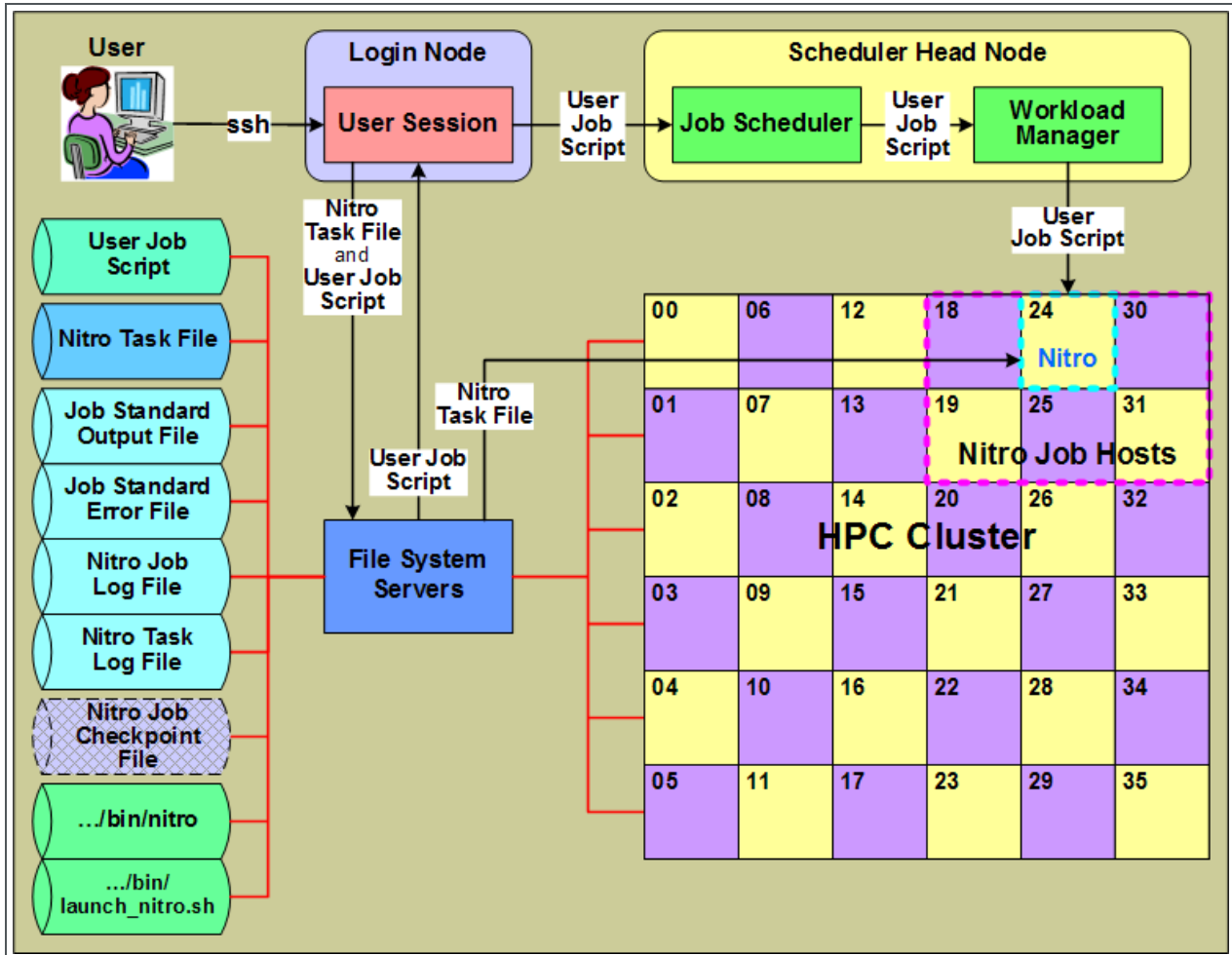
A Nitro job is a user job script that defines where the Nitro task file is and launches the Nitro application. The user job script can tailor the Nitro application's execution through the use of pre-defined Nitro environment variables.

When executed, the Nitro application takes the information from the Nitro task file and generates output to four files, the Nitro Job Log File and Nitro Task Log File, and the job's Standard Output File and Standard Error File.

- The Nitro Job Log File contains information about the Nitro job and the resources it used, a summary of all the tasks' status, and performance information. This is information you will want to look at to determine quickly how well your tasks executed and Nitro performed.

- The Nitro Task Log File contains information about the execution, status, and performance of each individual task; i.e., the results of the tasks.

- The job's Standard Output File contains information the Nitro application outputs in real-time about what it is doing. Some of the information in the Nitro Job Log File also appears here, but not all.

- The job's Standard Error File contains messages about any errors the Nitro application encountered.

In addition, while the Nitro job is executing, the Nitro application saves checkpoint information in a Nitro Job Checkpoint File that permits a Nitro job to restart later and resume where it was at because it was cancelled, preempted, etc. When the Nitro application finishes processing all tasks, it deletes the checkpoint file.

How to Create and Run a Nitro Job

A Nitro job is a user job script that defines where the Nitro task file is and launches the Nitro application.

You can create the task file on any system that permits you to create and edit a text file. (for example, Windows or Linux systems).

Typically, your administrator will give you a sample job script to use as the starting point for your user job script. You can pass your task file name/path to your Nitro job either by

- assigning it to a specific Nitro environment variable within the job script

- defining the Nitro environment variable before submitting the job script and then passing the environment variable and its value via the job submission command to the job script.

After creating the task file, and possibly customizing the job script, you submit your Nitro job script to your site's job scheduler to request a quantity of resources (for example, five hosts) on which to execute your Nitro job. The job scheduler schedules and allocates the requested resources to your job and then starts your Nitro job when the resources become available.

Your job script then executes the Nitro application via the launch_nitro.sh script and thehe Nitro application reads your task file and executes its tasks on your job's resources as quickly and with as little overhead as possible.

While tasks are executing, the Nitro application updates its Nitro Job Log File every five seconds to indicate its progress processing the tasks in your Nitro task file. In addition, for each task it processes, it outputs a record of the task, its status, and performance information, as well as possible task output, to the Nitro Task Log File.

When the Nitro job finishes,

- the Nitro Job Log File contains a final record of the Nitro job's information.

- the Nitro Task Log File contains a record for every task defined in the Nitro task file.

Factors Affecting Nitro Job Performance

You execution time will vary based on these factors:

- individual task execution times

- task resource requirements (e.g., single-core versus multi-core)

- quantity of resources you requested for your Nitro job

For example:

If a host has 16 cores, it can execute 16 single-core tasks at once (simultaneously); however it can only execute 2 multi-threaded tasks that require 8 cores each. You can expect a faster tasks per second rate with the first scenario (16 single-core tasks), than with the 2 multi-threaded tasks scenario.

Key Terminology and Usage

This document includes a glossary of key terms used through this guide. This is to help simplify and clarify the information presented.

For example, instead of using terms specific to the traditional HPC (research, university, and government institutions) and commercial enterprise markets, such as "HPC cluster" and "datacenter" and their corresponding "compute node" and "server" terms, this guide uses the generic terms "system" and "host", respectively. Also this guide uses the term "workload" to represent an arbitrary amount of work to execute on a system while "job" refers to workload submitted by a user to a system's scheduler for eventual execution on one or more of the system's hosts.

Refer often to for a complete list of terms used in this guide.

Chapter 2 Using Nitro

This chapter provides information and instructions on using Nitro.

In this chapter:

Prepare a Nitro Job

This topic provides information on the Nitro job's task file and performance tuning information.

In this topic:

Task File

The task file is a single file that contains a list of tasks to execute. Each line of the task file should contain only *one* task. You can add comments to your task file to help describe the tasks being performed, the data required, or other information that is pertinent to describing the tasks. Nitro also provides the capability to use task names and labels to help you organize your tasks.

Most of the tasks you create for a task file will probably run to completion fairly quickly, but it is possible that a task gets stuck in a loop or needs to run for a certain amount of time. Nitro by default limits tasks to 3,600 seconds (1 hour), but you can specify the limit to apply to the task by using the "maxtime" token. Time limits are specified in seconds. The following is an example of a task definition that limits a task to 30 seconds.

```
name=S23T01 maxtime=30 cmd=/opt/framemaker/bin/framegen -i /shared/scene23.def -tindex 1
```

Tasks

A task line can be as simple as the command you want to execute. For example, if you want to run a program called "framegen", input a file from a shared directory, and process the frame starting at time index "0" (zero), the command line might look like as follows.

```
/opt/framemaker/bin/framegen -i /shared/scene23.def -tindex 0
```

Nitro uses name/value pairs before the command line that you want to execute to define Nitro-specific information, such as, specifying a task name, task labels (that you can use to categorize the task), maximum time a task will run, and the command to execute to run the tasks itself. The key words for these name/value pairs are:

```
"cores=<count>"
"env=<name=value>[,<name=value>,...]"
"labels=<label>[,<label>,...]"
"name=<task name>"
"maxtime=<time limit in seconds>"
"memory=<amount>"
"shell=[default | none | <shell path>]"
"cmd=<command line>"
```

> ℹ The optional name/value pairs must be prepended to the line containing the task command line. As soon as Nitro sees something that isn't a name/value pair, the task line parsing stops and the rest is assumed to be part of the command line to execute.
>
> To make it clear where the task options end and your command line begins, include "cmd=" before your task's command line. This token is optional but helps to make the task definition easier to read when you are specifying other options. The following is an example command line with the "cmd=" token.
>
> ```
> cmd=/opt/framemaker/bin/framegen -i /shared/scene23.def -tindex 0
> ```

Nitro organizes the tasks for tracking.

Nitro tracks tasks by a task ID and line number. Nitro automatically generates a task ID for each task definition in the task file. The first task definition receives task ID "1".

> ℹ Only a task definition will increment the task ID. Because a task file can have empty or comment lines, the task ID and the line number in the task file may not be the same for the task.

Nitro will create a report of all tasks run and will include the task ID and line number in this report. The task ID is passed to the task via the **$NITROTASKID** environment variable.

To make Nitro tasks easier to track, or to search for specific tasks in the task completion report, add a unique task name to your task definition. Task names don't have to be unique, but creating a unique task name helps you identify specific tasks.

> ℹ You can use any naming scheme you want, as long as the name does not include spaces (which would indicate an end to the name/value pair).

For example, if you are processing data for scenes 21, 22, and 23, you can name the tasks according to scene and time index.

```
name=S21T00 cmd=/opt/framemaker/bin/framegen -i /shared/scene21.def -tindex 0
name=S21T01 cmd=/opt/framemaker/bin/framegen -i /shared/scene21.def -tindex 1
name=S22T00 cmd=/opt/framemaker/bin/framegen -i /shared/scene22.def -tindex 0
name=S22T01 cmd=/opt/framemaker/bin/framegen -i /shared/scene22.def -tindex 1
name=S23T00 cmd=/opt/framemaker/bin/framegen -i /shared/scene23.def -tindex 0
name=S23T01 cmd=/opt/framemaker/bin/framegen -i /shared/scene23.def -tindex 1
```

Nitro makes the task name available to the task via the **$NITROTASKNAME** environment variable when it executes the task. If the task command line includes the environment variable, it is substituted by its value before the command executes.

You can also use task labels to organize or identify the tasks. You can use multiple labels to describe a task. Multiple label values are separated by a comma between them; spaces are not allowed.

For example, if scene 22 contains a green screen that needs additional processing after this job completes, you can include the label "green" on all of the tasks for this scene.

```
name=S21T00 cmd=/opt/framemaker/bin/framegen -i /shared/scene21.def -tindex 0
name=S21T01 cmd=/opt/framemaker/bin/framegen -i /shared/scene21.def -tindex 1
name=S22T00 labels=green /opt/framemaker/bin/framegen -i /shared/scene22.def -tindex 0
name=S22T01 labels=green /opt/framemaker/bin/framegen -i /shared/scene22.def -tindex 1
name=S23T00 cmd=/opt/framemaker/bin/framegen -i /shared/scene23.def -tindex 0
name=S23T01 cmd=/opt/framemaker/bin/framegen -i /shared/scene23.def -tindex 1
```

Performance Tuning

Assignment Size

Each set of tasks that a coordinator sends to a worker is called an assignment. Nitro is most efficient when it can send a large enough assignment to each worker to keep the worker busy for at least 10 seconds before requesting more work. On the other hand, if you have a heterogeneous set of hosts with a wide variance in performance characteristics, you don't want one worker taking a very long time completing its assignment after all of the other workers have finished.

> **ⓘ** You can specify an assignment size as small as "1" (which could be useful for tasks that need to use all of the available OS cores) and as large as "1000" (useful to keep worker cores busy with tasks of extremely short duration).

> **ⓘ** Assignment sizes don't need to be evenly divisible by the number of OS cores available. Nitro will try to send the worker a second assignment when the worker gets about half way done with the current assignment so the second assignment will start running tasks as soon as an OS core becomes idle from the previous assignment.

Nitro uses a default assignment size of 250 tasks. If this is too many or two few tasks per assignment for your configuration, you can change it by using the --assignment-size command line option for the coordinator.

For example, if your nodes are all running 16 OS cores and each task takes 2 seconds to complete, each assignment of default size will take 31.25 seconds to complete (250 tasks at 2 seconds each divided by 16 OS cores), so you might want to change the assignment size to "80" to get an assignment time of closer to 10 seconds with the command line option.

> **ⓘ** Adaptive Computing recommends a 10-20 second assignment duration to optimize node utilization and to prevent "tailing" jobs (job where at its end there is only one or a few workers executing a large assignment and other workers are idle).

```
--assignment-size 80
```

If submitting a job to a job scheduler, you can change the assignment size by setting the **NITRO_COORD_OPTIONS** environment variable so it contains the --assignment-size command line option.

Thread Control

Nitro typically runs one task per available OS core on each worker. However, you can configure Nitro to run more tasks than OS cores (over-subscription), fewer tasks than OS cores (under-subscription) or a specific number of OS cores. You might want to over-subscribe the available OS cores if you are not utilizing the full capacity of the host. You may need to under-subscribe cores if

background tasks are running on the nodes. To over- or under-subscribe, use the --thread-ratio command line option.

```
--thread-ratio <ratio>
```

> *<ratio> only applies to worker nodes. However, if you are using the --run-local-worker command line option, then the thread ratio will be passed on to the coordinator's local worker.*

--thread-ratio also lets you specify over- or under-subscription properly in a heterogeneous host environment where hosts have different numbers of processors, cores, or threads. For example, if your hosts are all single socket, oct-core with hyper-threading enabled (16 total OS cores), but you want to over-subscribe by a factor of 1.5x, you could accomplish this by adding "--thread-ratio 1.5" to the worker command line to give each worker the ability to run 24 concurrent tasks. Alternately, if your tasks are all designed to use 2 OS cores each (multi-threaded application), you could use "--thread-ratio 0.5".

There may also be cases where you want to specify the *exact* number of OS cores to be used by the worker, such as when you have tasks that will use all available OS cores. In that case, you would use the --thread-count command line option to specify a thread count of 1 (--thread-count 1).

Run a Worker on the Coordinator Node

In configurations where you will be running less than 20 worker hosts, the coordinator node may be underutilized. To remedy this situation you may want to run a worker on the coordinator host so you can use its resources more effectively. To run a worker on the coordinator host, include the --run-local-worker flag on the coordinator's command line *or* you can explicitly start a worker Nitro process on the host.

> ⓘ Nitro will calculate the number of threads that the local worker should run so the coordinator is not starved for CPU cycles; causing it to slow down all the other workers.

Task Execution Environment Variables

Nitro will pass several environment variables to your tasks when it executes them.

Valid environment variables:

- $NITROJOBID – Job ID of the Nitro job.

- $NITROJOBDIR – Job directory to which Nitro writes log files. This directory can be used to store output files from your tasks.

- $NITROTASKCORES – Number of cores allocated to the task.

- $NITROTASKID – Task ID of the task. The task ID is a number that starts at 1 and increments by 1 for each task definition (valid or invalid) in the task file. Commented and empty lines are not counted; if the task file contains such, the task ID and the line number will diverge.

- $NITROTASKMEMORY – Amount of memory (in MB) allocated to the task.

- $NITROTASKNAME – Task name, if provided by the task definition.

- $NITROTASKTIME – Task time limit, specified by "maxtime" in the task definition.

- $NITRO_TASK_FILE - Can be used with normal file names that do not use spaces, but **MUST** be used if the user submits more than one task file.

- $NITRO_LONG_TASK_FILE - Can be used with normal file names that do contain spaces, but **MUST** be used if the file name contains spaces. This variable can only contain one file name. You cannot submit multiple file names containing spaces.

> ⚠️ Nitro reads a portion of the task file at a time. While the Nitro job is running, do *not* add or remove any task definitions or comment lines in the task file. Changes to the task file could cause line numbers to be changed and jobs to not run or be accidentally rerun.

Related Topics

Submit a Nitro Job to a Scheduler

This topic provides information about submitting a Nitro job and executing Nitro. The user creates the user job script and the system administrator sets up and configures the Nitro launch script, which the user job script executes to run Nitro.

Adaptive Computing's objective for Nitro is two-fold; first, allow the user to use the same job script for a Nitro job regardless of the scheduler used by a system to schedule jobs and, second, allow the administrator to customize Nitro for the system based on the scheduler and/or resource manager the system uses. This means the user job script can be scheduler-agnostic while the Nitro launch script (executed by the user job script) must be scheduler- or resource manager-specific.

In this topic:

User Job Script

Typically, the system administrator provides a sample Nitro job script to users to customize as needed.

Users can set launch script environment variables at the user job script-level to affect Nitro's behavior. See Launch Script Environment Variables on page 13 for a list of available variables.

The user job script also contains the information to execute the Nitro launch script (`/opt/nitro/bin/launch_nitro.sh`).

Launch Script Environment Variables

The Nitro launch script looks for several environment variables. Users can also customize these variables in the user job script.

- NITRO_TASK_FILE
- NITROJOBID
- NITRO_OPTIONS

- NITRO_COORD_OPTIONS
- NITRO_WORKER_OPTIONS.

NITRO_TASK_FILE

The NITRO_TASK_FILE environment variable must contain the path to the task file, and the task file must be accessible to the coordinator host that will run the job.

Users should put the task file in a shared location accessible by any host in the system.

NITROJOBID

The NITROJOBID environment variable is created automatically by the launch script, based on the resource manager's job ID, if the user job script does not provide one.

Unless you are restarting a job that partially completed and was canceled, you don't need to set this environment variable. If you specify this environment variable in the job's submission, it will override the resource manager job ID and Nitro will use the value you supplied.

> 🛈 If your job scheduler and resource manager use different numbering systems, the job ID that Nitro will use is the one that it gets from the resource manager. You may want to submit the job directly to the resource manager in this case to avoid confusion. Check with your system administrator to find out if your job scheduler's and resource manager's job ids are synchronized.

NITRO_OPTIONS, NITRO_COORD_OPTIONS, or _NITRO_WORKER_OPTIONS

Any command line options the user job script needs to pass to Nitro must be contained in either the NITRO_OPTIONS, NITRO_COORD_OPTIONS, or NITRO_WORKER_OPTIONS environment variables.

Moab/Torque Customization Commands and Options

If using Moab/Torque, be aware of the following:

- Nitro will consume all available OS cores on the hosts on which it runs.
  - If Nitro is running exclusively on hosts, you may want to configure your system to run a single job per host (for example, in Moab, set NODEACCESSPOLICY to SINGLEJOB in the moab.cfg file).
  - If you will run other workload on the system that may run multiple jobs per host, you will need to provide a script or instruct users on how to specify their Nitro jobs to run with a single task per host configuration. In Moab and Torque the jobs should be submitted (using either msub or qsub) with the "`-l nodes=<node count>:tpn=1`" command line option.

Examples

The sample launch_nitro.sh job scripts provided in `/opt/nitro/scripts/<resource manager>/` all expect environment variables. You can set environment variables for the Nitro launch script at job submission or define them in the user job script.

Example: Setting Environment Variables at Job Submission

The "-v" option in Moab's msub command sets environment variables before calling the user job script. The user job script inherits these environment variables.

> ℹ The user job script can add to or overwrite any environment variable defined in msub's "-v" option.

- User job script (/home/jdoe/user_job_script.sh)

  This script must exist on the host where you execute the msub command. In this example, this user job script exists in jdoe's home directory.

  ```
  exec /opt/nitro/bin/launch_nitro.sh
  ```

- msub example

  ```
  msub -lnodes=5 -ltpn=1 -lwalltime=600 -v "NITRO_TASK_
  FILE=/home/jdoe/nitro/monte_sim.txt,NITRO_OPTIONS=--job-dir
  /home/jdoe/monte01,NITRO_COORD_OPTIONS=--run-local-worker" /home/jdoe/user_job_
  script.sh
  ```

  - The task file monte_sim.txt contains the list of task command lines that you want to execute.

  - The "--job-dir" sets the path to which Nitro will write the job and task logs.

  - The last parameter is the path of the user job script.

Example: Setting Environment Variables In the User Job Script

- User job script (/home/jdoe/user_job_script.sh)

  This script must exist on the host where you execute the msub command. In this example, this user job script exists in jdoe's home directory.

  ```
  NITRO_TASK_FILE=/home/jdoe/nitro/monte_sim.txt
  NITRO_OPTIONS="--job-dir /home/jdoe/monte01"
  NITRO_COORD_OPTIONS=--run-local-worker
  exec /opt/nitro/bin/launch_nitro.sh
  ```

- msub example

  ```
  msub -lnodes=5 -ltpn=1 -lwalltime=600 /home/jdoe/user_job_script.sh
  ```

  - The task file monte_sim.txt contains the list of task command lines that you want to execute.

  - The "--job-dir" sets the path to which Nitro will write the job and task logs.

  - The last parameter is the path of the user job script.

  - NITRO_TASK_FILE, NITRO_OPTIONS and NITRO_COORD_OPTIONS are set in the user job script.

Track Job Progress

This topic provides information on viewing job progress and output.

In this topic:

Introduction on How Nitro Tracks the Job

Nitro will print some job information to stdout, such as what workers attached, how many tasks have been run, if any tasks failed, etc.

If your Nitro job is submitted through a scheduler, you may not see any of this until the job has completed and the resource manager has copied the job output to your job's submission directory.

However, Nitro provides a tool called nitrostat to display status information while the job is running. nitrostat is located in the `nitro/bin` directory where Nitro was installed.

Nitro creates two files that you can use to stay up-to-date on the progress of your job.

- nitro_<*jobid*>.joblog.txt - Information about the job in general.

- nitro_<*jobid*>.tasklog.txt - Listing of individual tasks that have completed along with performance statistics collected from running the task (duration and memory usage) and the task output to stdout and/or stderr.

> ⓘ Both files are written to the job directory that you provide using the `--job-dir` command line option when submitting your job, or to the default job directory `$HOME/nitro/<jobid>`.

Job Log

To see job status using nitrostat, you will need the job ID. The job ID is the job ID reported to you when you submitted the job to the schuleder *or* that you set manually via the `--job-id` command line option in the NITRO_OPTIONS environment variable or via the NITROJOBID environment variable.

- The default location for the job and task logs are in your "`$HOME/nitro/<jobID>`" directory.

- You can also use the "--job-dir" command line option to specify a different job directory if you are not using the default location.

Nitro Job Progress Report

The Nitro job progress reports lets you see the current contents of a job log file.

For example, let's say you have a job that was run by your resource manager as job "23576", running "/opt/nitro/bin/nitrostat 23576"shows you the job's progress.

```
Nitro Job Progress Report

Start Time  : 2016-02-10 09:10:11-0600
Current Time: 2016-02-10 09:10:42-0600
Elapsed Time: 31 seconds (00:00:31)

Job Id      : 23576
Coordinator : node01
Task Log    : /home/jdoe/jobs/23576/nitro_23576.tasklog.txt
Task File   : /home/jdoe/jobs/survey03.tasks
  File Size : 123366
  Est Tasks : 3016
  Processed : 75%

Tasks
------------
Pending     : 500
In Progress : 500
Completed   : 1250
  Success   : 1250
  Failure   : 0
  InsufRes  : 0
  Timeout   : 0
  Invalid   : 0
  Tasks/sec : 40.3
Total Tasks : 2250

Workers
-------
Host    Pid     Thrds Status  Assigned Running Completed  Success  Failure   InsufRes
Timeout Tasks/sec AsgmtDur
node02 6851     12  running     1250     250      1000      1000       0         0
    0       36.0      8.0
node03 14988     4  running      500     250       250       250       0         0
    0        9.3     27.0
```

The following describes the fields and their output descriptions.

- Start Time – Date and time the coordinator started running.

- Current Time – Current date and time the report was generated (reports are generated every 5 seconds).

- Elapsed Time – Amount of time the coordinator has been working on the tasks.

- Job Id – Job ID that Nitro was passed on its command line. Typically assigned by the resource manager, but can be assigned by the user.

- Coordinator – Host name on which the coordinator is running.

- Task Log – Path and file name of the task log file that is generated by the coordinator.

- Task File – Path and file name of the task file.

- File Size – File size of the task file.

- Est Tasks – Number of tasks the coordinator estimates in the task file. Since Nitro doesn't read the entire task file on startup, an estimate is given based on lines read from the file so far.

- Processed– Percentage of the task file that has been read by the coordinator.

- Tasks Section: Lists the counts of tasks in each category

  ○ Pending – Number of tasks that have been put into assignments and are waiting to be sent to a worker.

  ○ In Progress– Number of tasks in assignments sent to the workers for which workers have not yet returned results.

  ○ Completed – Number of tasks in assignments that have been completed (workers have returned results).

  ○ Success – Number of completed tasks that were successful (the task returned an exit code of 0).

  ○ Failure – Number of completed tasks that returned an exit code other than 0.

  ○ InsufRes – Number of tasks that could not be run because the requested resources for the task was not available.

  ○ Timeout – Number of completed tasks that ran longer than the task "maxtime" parameter and were terminated by the worker.

  ○ Invalid – Number of task definitions that contained errors and could not be run.

  ○ Tasks/sec – Number of tasks per second based on the time that the coordinator sends the first assignment until the time the report is generated. If in *linger mode*, this will only be calculated for the last 60 seconds.

  ○ Total Tasks – Total number of tasks including completed and invalid tasks.

- Workers Section: List by worker

  ○ Host – Host name and port (if not the default port) of the worker.

  ○ Pid – Process ID of the worker.

  ○ Thrds –Number of task launch threads the worker is using to run tasks.

  ○ Status – Status of the worker. This may be "unconnected", "running", "unresponsive", "closing", or "closed".

  ○ Assigned – Number of tasks assigned to this worker so far.

  ○ Running – Number of tasks in assignments currently allocated to the worker.

  ○ Completed – Number of tasks in assignments the worker has completed.

  ○ Success – Number of successfully completed tasks.

  ○ Failure – Number of tasks that returned an exit code other than 0.

  ○ InsufRes – Number of tasks that could not be run because the requested resources for the task was not available.

  ○ Timeout – Number of tasks that exceeded the tasks "maxtime" threshold and were terminated by the worker.

  ○ Tasks/sec – Number of tasks per second that the worker has completed so far. In

> *linger mode* this is only calculated for the last 60 seconds.

- ○ AsgmtDur – Average assignment duration in seconds.

Job Completed Report

Once the job has completed, the job report will show "(final)" on the end of the first line of the report and Current Time is replaced with Finish Time (after Start Time). The following example is based on the previous example for job "23576" .

```
Nitro Job Progress Report (final)

Start Time  : 2016-02-10 09:10:11-0600
Finish Time : 2016-02-10 09:11:36-0600
Elapsed Time: 85 seconds (00:01:25)

Job Id      : 23576
Task Log    : /home/jdoe/jobs/23576/nitro_23576.tasklog.txt
Task File   : /home/jdoe/jobs/survey03.tasks

Tasks
-------------
Pending    : 0
Running    : 0
Completed  : 3000
  Success  : 3000
  Failure  : 0
  InsufRes : 0
  Timeout  : 0
  Invalid  : 0
  Tasks/sec : 35.3
Total Tasks : 3000

Coordinator
-----------
Host    : node01
Threads : 8

Worker Resources
----------------
Workers : 2
Threads : 16

Workers
-------
Host    Pid     Thrds Status Assigned Running Completed  Success  Failure  InsufRes
Timeout Tasks/sec AsgmtDur
node02 6851     12  closed    2250       0     2250      2250       0         0
  0      29.2       8.3
node03 14988     4  closed     750       0      750       750       0         0
  0       8.8      35.7
```

Task Log

The task log file contains a listing of all tasks that have been completed and some statistics about the tasks duration and memory consumption. This file is named `nitro_<JobID>.tasklog.txt` and is located in the same directory as the job log file.

The task log file is tab-delimited, so you can easily import it into a spreadsheet or database, or process it using another program. You can also view the task log using the nitrostat utility.

```
JobID  TaskID  Line Name          Status  ExitCode Hostname           StartTime
         Duration UserCPU SystemCPU VirtualMem PhysicalMem Labels
Output
foo    1       1    task001      Success 0        localhost:10004    2015-06-18_
15:26:52.954-0600    1.005   0.000     0.000    7364608      630784
foo,foobar,foobaz,xyz
foo    2       2    task002      Success 0        localhost:10004    2015-06-18_
15:26:52.954-0600          1.007   0.000     0.000    87834368       630784 foo,foobar,xyz
foo    3       3    task003      Success 0        localhost:10004    2015-06-18_
15:26:52.954-0600          1.005   0.000     0.000    71728640       901120 foo,xyz
foo    4       4    task004      Success 0        localhost:10004    2015-06-18_
15:26:52.955-0600    1.005   0.000     0.000    38837504     630784
foo,foobar,foobaz,abc
foo    5       5    task005      Success 0        localhost:10004    2015-06-18_
15:26:53.960-0600          1.004   0.000     0.000    405946368      630784 foo,foobar,abc
foo    6       6    task006      Success 0        localhost:10004    2015-06-18_
15:26:53.961-0600    1.005   0.000     0.000  405946368      946176 foo,abc
foo    7       7    task007      Success 0        localhost:10004    2015-06-18_
15:26:53.961-0600          1.003   0.000     0.000  405946368      630784
foo    8       8    task008      Success 0        localhost:10004    2015-06-18_
15:26:53.966-0600          1.003   0.000     0.000  405946368      700416
foo    9       9    task009      Success 0        localhost:10004    2015-06-18_
15:26:54.965-0600          1.005   0.000     0.000  405946368      630784
foo    10      10   task010      Success 0        localhost:10004    2015-06-18_
15:26:54.965-0600          1.003   0.000     0.000  405946368      630784
foo    11      11   task011      Success 0        localhost:10004    2015-06-18_
15:26:55.973-0600          1.005   0.000     0.000    7364608      630784
foo    12      12                Success 0        localhost:10004    2015-06-18_
15:26:55.973-0600          1.004   0.000     0.000  405946368      626688
foo    13      14   fail         Failure 1        localhost:10004    2015-06-18_
15:26:55.973-0600          0.005   0.000     0.000      8192        4096
foo    14      16   stderr       Success 0        localhost:10004    2015-06-18_
15:26:55.974-0600          0.005   0.000     0.000  405946368      536576
foo    15      18   stderr_fail Failure 1        localhost:10004    2015-06-18_
15:26:55.979-0600          0.005   0.000     0.000  405946368     1228800
        ERROR MESSAGE
foo    16      20   overtime    Timeout -9        localhost:10004    2015-06-18_
15:26:55.980-0600          2.006   0.000     0.000  405946368      970752
        maxtime exceeded, process was killed
foo    17      21                Success 0        localhost:10004    2015-06-18_
15:26:55.985-0600          1.002   0.000     0.000  405946368      626688
foo    19      23                Success 0        localhost:10004    2015-06-18_
15:26:56.979-0600          1.007   0.000     0.000  405946368      970752
foo    20      24                Success 0        localhost:10004    2015-06-18_
15:26:56.988-0600          1.003   0.000     0.000  405946368      724992
foo    21      25                Success 0        localhost:10004    2015-06-18_
15:26:57.986-0600          1.005   0.000     0.000  405946368      724992
foo    22      26                Success 0        localhost:10004    2015-06-18_
15:26:57.988-0600          1.005   0.000     0.000  405946368      970752
foo    23      27                Success 0        localhost:10004    2015-06-18_
15:26:57.988-0600          1.005   0.000     0.000  405946368      630784
foo    24      28                Success 0        localhost:10004    2015-06-18_
15:26:57.995-0600          1.005   0.000     0.000  405946368      630784
foo    25      29                Success 0        localhost:10004    2015-06-18_
15:26:58.993-0600          1.005   0.000     0.000  405946368      974848
foo    26      30                Success 0        localhost:10004    2015-06-18_
15:26:58.994-0600          1.004   0.000     0.000  405946368      626688
```

The task log contains the following fields.

- JobID – Job ID that was passed to Nitro using the "--job-id" command line option.

- TaskID – Task number within the Nitro job.

- Line – Line number in the task file of the task definition.

- Name – Task name supplied in the task definition by the "name=*<name>*" option.

- Status – One of "Success", "Failure", "InsufRes", "Timeout", or "Invalid".

- ExitCode – Numerical exit code returned by the task.

- Hostname – Name of the worker that executed the task.

- StartTime – Date and time the worker actually started the task.

- Duration – Number of seconds the task ran (millisecond resolution).

- UserCPU – Number of seconds the task ran in user mode (millisecond resolution).

- SystemCPU – Number of seconds the task run system calls (millisecond resolution).

- VirtualMem – Maximum virtual memory allocated to the task in bytes.

  > ⓘ The operating system may allocate shared memory and may charge a proportion of this shared memory to random tasks.

- PhysicalMem – Maximum physical memory allocated to the task in bytes.

- Labels – Optional task labels specified by the task definition.

- Output – stdout and/or stderr. If a task outputs to both stdout and stderr, both are displayed in the format *<stdout>*/*<stderr>*.

Related Topics

Chapter 3 References

This chapter provides additional information for system administrators and users.

In this chapter:

Command Line Options

This topic identifies the individual command lineoptions recognized and/or required by Nitro.

In this topic:

Options

- **Thread Count** – The quantity of threads the Nitro workers should use when executing tasks. This option is mutually-exclusive with the Thread Ratio option.

  If this option and the Thread Ratio option are not given, a worker uses one task launch thread per OS core to which it is pinned.

  The primary reason for this option is to explicitly specify a task-launch thread count for Nitro running a specific single application, usually on homogeneous hosts.

  ```
  --thread-count <num>
  ```

- **Thread Ratio** – The ratio of task launch threads-to-OS "cores" the Nitro workers should use when creating task launch threads. This option is mutually exclusive with the Thread Count option.

  If this option and the Thread Count option are not given, the ratio is "1.0", meaning a worker uses one task launch thread per OS core to which it is pinned.

  Ratio is a positive real number (e.g., 1.5, 0.5, etc) that when multiplied with the count of OS cores to which a worker is pinned yields a count of the task launch threads it will use. The worker rounds the count to the nearest integer, with a minimum value of 1.

The primary reason for this option is to allow a user to over-subscribe or under-subscribe the task-launch thread count appropriately relative to the OS core count of heterogeneous hosts (e.g., 1.5 means 6 threads for a quad-core host and 24 threads for a 16-core host).

```
--thread-ratio <ratio>
```

- **Assignment Size** – The quantity of tasks the Nitro coordinator should pass to a Nitro worker at one time; default is 250.

```
--assignment-size <num>
```

- **Job Directory** – Specifies the path for the directory where Nitro will place its Job Progress Log and Completed Task Log files.

```
--job-dir <path>
```

- **Job ID** – Specifies the job ID for a specific Nitro run. The job ID may be used to create the job directory and certain file paths.

```
--job-id <jobID>
```

- **Coordinator Threads** – Indicates to the coordinator how many threads to reserve for the coordinator when allocating cores to a local worker (when using "--run-local-worker" on the coordinator command line). Default is 2. Adaptive Computing recommends setting the *<count>* value to 1 if all jobs will use less than 20 nodes and setting the *<count>* value to 4 if the jobs require a large number of nodes (greater than 50) to run.

> 🛈 This option overrides the --coord-threads setting in the nitro.cfg file.

```
--coord-threads <count>
```

- **Task Environment** – Specifies the environment variables to set in the task's execution environment. This is used by the worker but is also needed on the coordinator's command line if running a local worker. Multiple values can be specified by separating name/value pairs with a comma.

```
--task-env <ENVVARNAME=value[,...]>
```

Command Line Options per Nitro Mode

The table that follows identifies which command line options Nitro uses in worker or coordinator mode. Some command line options are used in both modes and are listed in this table in the "Both" row.

Nitro Mode	Command Line Option
Coordinator	--assignment-size --coord-threads (if using --run-local-worker with the coordinator)

Nitro Mode	Command Line Option
Worker	--task-env (if *not* using --run-local-worker with the coordinator) --thread-count (if *not* using --run-local-worker with the coordinator) --thread-ratio (if *not* using --run-local-worker with the coordinator)
Both	--job-dir --job-id --task-env (if using --run-local-worker with the coordinator) --thread-count (if using --run-local-worker with the coordinator) --thread-ratio (if using --run-local-worker with the coordinator)

Task File

A task file contains a list of Nitro task definitions (task execution options) along with the task command line Nitro will execute. Since the Nitro coordinator will be running on one of the nodes allocated to the Nitro job, the task file must be accessible to the node on which the coordinator will run.

The task file is a text file where each task definition must be contained on a single line. Lines of text may be terminated by either a Linux-style line ending (LF or '\n' new line character) or a Windows-style line ending (CR/LF - '\r\n' carriage return/line feed combination). The line number is reported in the task log so that errors in the task file can be quickly located and fixed.

The task file allows comment and empty lines. A hash symbol (#) in the first column of a line identifies a comment line.

Each task will be assigned a task ID, which will start at 1 and increment with each task line (comment and empty lines are not assigned a task ID). This task ID is passed to the task in the NITROTASKID environment variable.

Task Options

Task options are name/value pairs that are listed before the task's command line of the form "*<option>*=*<value>*". Task options must be specified *before* the task's command line to be executed. As Nitro parses the line, it will stop looking for name/value pairs as soon as it finds a character string that does not include the name/value delimiter (=) or is the "cmd" option. Everything after the "cmd=" option or the first string that is not delimited as a name/value pair will be considered part of the task command line.

Task definitions that contain errors (such as a misspelled option) are considered "invalid" tasks and will be reported in the task log along with an explanation of the error in the line. Examples of valid command lines are as follows:

```
# Commented line
/opt/framemaker/bin/assemble_frame --input /shared/scene23.def --time-index 0
cmd=/opt/framemaker/bin/assemble_frame --input /shared/scene23.def --time-index 0

name=Scene23Time0 /opt/framemaker/bin/assemble_frame --input /shared/scene23.def --
time-index 0
name=Scene23Time0 maxtime=30 cmd=/opt/framemaker/bin/assemble_frame --input
/shared/scene23.def --time-index 0
```

The following describes the various task options.

- **Application Command** – The coordinator considers everything immediately after the equal sign (= in "cmd=") as the task's "application" command line, which a worker will execute. There must be at least one non-whitespace character immediately after the "=" or the coordinator declares the task definition invalid. The application command line permits standard I/O redirection and environment variable substitution.

  ```
  cmd=<xxx -y zzz>
  ```

  > ⚠ Do not place any task options after the command line or the coordinator will not parse them; assumes they are part of the command line.

- **Labels** – Specifies the labels assigned to a task.

  This is optional and there is no default value.

  If given, a label must be composed of letters, digits, underscore, hyphen, and/or period. Use a comma to separate multiple labels.

  If the option's value violates the conditions above, the coordinator will declare the task definition invalid and will not send the task to a worker.

  When the coordinator logs the task in the Completed Tasks Log file, it outputs this option's value "as is", meaning without alternation and with no substitution of commas with spaces.

  ```
  labels=<list>
  ```

- **Maximum Time** – Maximum time (in seconds) a task may execute after which the worker will terminate it. This is optional; the default value is 3,600 seconds (1 hour).

  If given, the value must be less than the `maxtime-limit <period>` value.

  > ℹ If the option's value is non-numeric, non-decimal, or outside the allowed range, the coordinator will declare the task definition invalid and will not send the task to a worker.

  ```
  maxtime=<nn>
  ```

- **Name** – Unique name assigned to your task definition. Task names do not have to be unique, but creating a unique task name will help to identify tasks.

  ```
  name=<task name>
  ```

- **Task Cores** – Number of OS cores that the task requires. Nitro will allocate the number of cores requested and set the affinity of the task to the available cores.

```
cores=<count>
```

> ⚠️ The command line options "--thread-count" or "--thread-ratio" affect the number of available cores. If you use either of these options and they specify more cores than the node has available, Nitro will not pin the task to a specific core. If a task is specified to require more cores than the node that receives the task assignment, the task will not run.

> ℹ️ Users should submit their jobs so that tasks can run on any of the nodes that are allocated to the job. If, for example, you have some tasks that require 20 processors, but there are some 16 core nodes in the cluster, the job should be submitted so that it only allows 20 proc nodes to be allocated to the job. If a Nitro worker is assigned a job with requirements that it cannot fulfill (either too many cores, or too much memory) the task will be counted as failed, and Nitro will show the status of "InsufRes" for that task in the task log file.

- **Task Environment Variables** – Specifies a list of user supplied environment variables that will be set in the context of the task. The list of environment variables can be one or more name/value pairs separated by commas. Environment variable name value pairs cannot contain spaces.

```
env=<name=value>[,<name=value>,...]
```

- **Task Memory** – Maximum amount of memory that the task requires. Nitro determines the amount of physical memory available on the system and uses this number as the limit that can be allocated by concurrent tasks. If no units are specified, GB is assumed. Available unit specifications include "GB" (10^9 bytes), "GiB" (2^{30} bytes), "MB" (10^6 bytes), and "MiB" (2^{20} bytes). Nitro uses MB units in debug logs.

```
memory=<amount>
```

> ℹ️ Users should submit their jobs so that tasks can run on any of the nodes that are allocated to the job. If, for example, you have some tasks that require 32 GB, but there are some 16 GB nodes in the cluster, the job should be submitted so that it only allows 32 GB nodes to be allocated to the job. If a Nitro worker is assigned a job with requirements that it cannot fulfill (either too many cores, or too much memory) the task will be counted as failed, and Nitro will show the status of "InsufRes" for that task in the task log file.

- **Task Shell** – Specifies the task shell, if any, to use. Tasks are normally executed by running "/bin/bash -c <*task command line*>".

```
shell=[default | none | <shell path>]
```

- The default shell provides translation of environment variables into command line options and other command line processing benefits.

- In high-throughput environments performance gains can be realized by using a lighter weight shell such as the Bourne shell or Korn shell.

- If no command line processing is needed, the task can be run without a shell.

- Executing a task directly instead of using the shell can speed task invocation by more than 50% over the default shell.

When specifying a shell other than the default shell, the fully qualified path should be used. For example, if you want to use the Bourne shell you should specify the shell as "/bin/sh" as opposed to just "sh".

nitrostat

nitrostat is a utility found in the `/opt/nitro/bin` directory that will display the status of a Nitro job or of individual tasks. nitrostat lets you quickly find specific tasks or list all failed, invalid, or timed out tasks. nitrostat also offers a "wait" mode that will monitor the task log for tasks matching the specified criteria until the job completes.

Running nitrostat

To run nitrostat, you'll need to know the job ID of the job you want to monitor. For example, if you have a job with ID "3145", you can monitor the job progress with the following command:

```
/opt/nitro/bin/nitrostat 3145 -w
```

nitrostat assumes that the job information can be found in `$HOME/nitro/<job id>`.

If you have specified a different location for the job directory using the Nitro "`--job-dir`" command line option, then you'll need to specify the same location using the nitrostat "`--job-dir`" command line option.

For example, if your job directory is in `$HOME/projects/survey03` then use the following command:

```
/opt/nitro/bin/nitrostat 3145 --job-dir $HOME/projects/survey03 -w
```

nitrostat will show the following information when job status is requested:

```
Nitro Job Progress Report

Start Time   : 2015-06-17 09:10:11-0600
Current Time: 2015-06-17 09:10:42-0600
Elapsed Time: 31 seconds (00:00:31)

Job Id       : 23576
Coordinator : node01
Task Log     : /home/jdoe/projects/survey03/23576/nitro_23576.tasklog.txt
Task File    : /home/jdoe/projects/survey03/survey03.tasks
  File Size  : 123366
  Est Tasks  : 3016
  Processed  : 75%

Tasks
------------
Pending    : 500
Running    : 500
Completed  : 1250
  Success  : 1250
  Failure  : 0
  InsufRes : 0
  Timeout  : 0
  Invalid  : 0
  Tasks/sec : 40.3
Total Tasks : 2250

Workers
-------
Host    Port  Pid     Thrds Status  Assigned Running Completed  Success  Failure
InsufRes Timeout Tasks/sec AsgmtDur
node02 47000 6851     12  running      1250     250      1000     1000        0
0       0      36.0     8.0
node03 47000 14988     4  running       500     250       250      250        0
0       0       9.3     27.0
```

Searching for Task Records

You can use nitrostat to search the task log by task name, task ID, or label using regular expressions. You can also combine criteria to further refine your search.

For example, if you want to search for tasks containing the task name "Survey03" and the label "NYC" you can specify the command line as follows:

```
/opt/nitro/bin/nitrostat Job01 --name Survey03 --label NYC
```

The following identifies the nitrostat command line options.

- `--all, -a` – Shows all tasks.

- `--completed, -c` – Shows completed tasks.

- `--failed, -f` – Shows failed tasks.

- `--invalid, -i` – Shows invalid tasks.

- `--timedout, -o` – Shows tasks that timed out (exceeded maxtime).

- `--wait, -w` – Continues updating results until entire job is completed.

- `--name, -n <task name>` – Shows task(s) with the specified task name.

- `--task, -t <task id>` – Shows the task with the specified task ID.

- `label,-l <label list>` – Shows all tasks that contain the specified label. *<label list>* is a comma-separated list of labels that may *not* contain spaces.

- `--working-dir,-d <directory>` — Uses the specified working directory to locate the job and task log files. The default working directory is `$HOME/nitro`.

- `--regex` – If set, uses regular expression as the matching mode for *<task name>*, *<task id>*, and *<label list>*. The default is literal (exact string).

Job Recovery

Jobs run under a scheduler can, depending on job priority and settings, be preempted by a higher priority job, or even canceled by the user or administrator, or may fail due to hardware failure. Depending on the scheduler's configuration, a preempted job may be restarted later by the scheduler using the same job ID as the original job.

The job ID is the key to recovering jobs since Nitro uses the job ID as part of the path to the files associated with that job. Nitro tracks its progress by storing a checkpoint file that indicates which tasks have been completed and which have not. When Nitro is restarted, it looks for a checkpoint file and will continue from where it left off if one is found. If a job was canceled or preempted without a restart policy, then you will need to restart the job manually. Again, the key to restarting the job is to use the job ID of the original job.

The job ID is usually the ID that was returned when the job was submitted. There can be some differences between the scheduler's job ID and the resource manager's job ID depending on scheduler and resource manager settings. When you submitted your Nitro job, you may have set a Nitro job directory. If you didn't, it defaults to `$HOME/nitro/<jobid>`. This directory will contain the job log and task log files, along with checkpoint and Nitro log files. You can therefore use the directory name that Nitro created as the job directory with which to resubmit the job by passing the `--job-dir` option with the directory name through the NITRO_OPTIONS environment variable.

To restart the job you must set the NITROJOBID environment variable to the original job ID. Setting this environment variable will override the job ID provided by the resource manager and Nitro will resume from the line number of the task file described in the checkpoint file.

The checkpoint file is updated periodically when assignments are completed by workers and are returned to the coordinator. If a job is canceled, the workers will do their best to respond to the coordinator with the tasks that have been completed so far, but depending on how quickly the resource manager forces the applications to close, the checkpoint file may or may not be fully updated. Therefore, it is possible that restarting a job will result in a particular task or set of tasks being run a second time. Users should take this into account and program their tasks so that if running the task a second time would cause a problem, transactions are recorded by the task that would prevent the second run.

If a job is canceled for reasons of task failure (for example, because of a typo in the task command line), you may want to submit the job as a new job instead of trying to resume the job with failed tasks.

> ⓘ Failed and invalid tasks are marked as complete in the checkpoint file; they won't be re-run if the job is just restarted.

Glossary

Compute Node

Term for a computer (see Server) designed for high-performance computing and managed by an HPC administrator as part of an HPC cluster (see Host)

Coordinator

Nitro component responsible for scheduling Nitro tasks to the Worker components for execution, recording the tasks' information in the Task Log file and job information in the Nitro Job Log file, and checkpointing the Nitro job's state information in the Nitro Checkpoint file.

Core

An individual hardware-based execution unit within a processor that can independently execute a software execution thread and maintain its execution state separate from the execution state of all other cores within the processor.

CPU

See Processor, Core, Thread, OS Core, and Virtual Core. CPU is too generic, ambiguous, or context-specific for utilization in this manual.

Datacenter

A non-HPC cluster system composed of many "servers" that typically are not used for high performance computing.

High Performance Computing

The use of highly parallel and/or specialized "supercomputers" for executing parallel workloads such as large simulations, solving problems that require very complex and extensive calculations, computations that require very long running calculations, etc. Such workloads are characterized by their use of many "compute nodes" (servers or hosts), often in the thousands, to work on a single problem and have execution times ranging from minutes to months. HPC systems often execute from one to a few dozen or hundreds of simultaneous workloads and have a job (workload) queue with a few hundred to several thousands of pending jobs. In HPC systems, the performance of individual workloads within a time interval is the primary objective and therefore HPC schedulers attempt to optimize their use of an HPC system's resources regardless of the scheduling overhead incurred to do so (within reason).

High Throughput Computing

Describes workloads that often execute on just a single core and may have execution times ranging from sub-seconds to minutes and perhaps hours. HTC systems often execute hundreds to tens of thousands of simultaneous workloads. In HTC systems, the quantity of workloads processed per time interval is the primary objective and therefore HTC schedulers attempt to minimize scheduling overhead in order to maximize workload throughput.

Host

Generic reference to an HPC system's "compute node" or a datacenter's "server".

HPC

See High Performance Computing.

HPC Cluser

HPC industry's term for a "supercomputer". It is somewhat analogous to a "datacenter", except for the sometimes specialized nature of its hardware.

HT
> See Hyper-Threading.

HTC
> See High Throughput Computing.

Hyper-Threading
> Term used by Intel for its Simultaneous Multi-Threading (SMT) capabillity in its Atom, Core, Itanium, Pentium 4, Xeon, and Xeon Phi processor families. See also Simultaneous Multi-Threading.

Job
> HPC term for workload submitted by a user to a scheduler for the purpose of scheduling resources on which the workload executes when started up by the scheduler. This document will use this term to identify workload, in whatever form, submitted to a scheduler that schedules the workload for execution on a system (HPC cluster or commercial datacenter). Typically, a user creates a script that executes the workload (one or more applications) and submits the script to the scheduler where it becomes a "job". The user also gives information identifying the types of resources, typically one or more hosts and optionally other hardware (such as GPU or MIC accelerators) or software and/or software licenses, required by the workload to execute, either in the job script itself or at the time of job submission (for example, via command line options or web portal form). The scheduler schedules the job for the requested resources and when they are available allocates them to the job and then starts the job by executing the script on one of the allocated hosts. The script executes the workload(s)/application(s) that then use the resources allocated to the job by the scheduler.

Job Scheduler
> HPC term for a scheduler that manages submitted workloads (called "jobs") for an HPC cluster. See Scheduler.

Multi-threading
> The use of multiple software threads, which may or may not be pinned to hardware threads (core affinity), to implement processing in parallel. There are multiple implementations of multi-threading, such as Linux "pthreads", etc.

Nitro
> HTC task scheduler application offered by Adaptive Computing, Inc.

Node
> Shorthand term for "compute node". See Compute Node.

OS Core
> Term that refers to what the operating system considers an individual hardware-based computation unit, often called a "core" or a 'CPU". In actuality, the "OS core" can be a hardware-based core (see "Core") or a hardware-based thread (see "Thread"). This guide uses this term to refer to the basic hardware-based computational unit allocatable by an operating system to a process.

Process
> An individual executing program managed by an operating system. It has its own resources and memory address space, independent of all other executing processes managed by the operating system. A process may itself be multi-threaded, which means the operating system can execute simultaneously different software execution threads of the process.

Processor

A physical hardware chip (sometimes called a "socket"); regardless of whether it supports a single core or multiple cores ("multi-core" processor). Socket is a strong, unambiguous synonym for processor while CPU (see CPU) is an ambiguous synonym individuals and/or processor vendor documentation may use. In addition, people and literature sometimes use the term processor to refer to a hardware core (see Core) or hardware thread (see Thread). This guide uses the term "processor" to refer to the physical hardware chip.

Scheduler

Term used generically in the guide for the specialized software between the user and the HPC cluster-/datacenter system that manages submitted workloads or "jobs". Such management includes queuing jobs, prioritizing queued jobs for execution, scheduling and allocating requested resources for each job, and starting jobs when their requested resources become available and the jobs have the highest priority. This guide uses the term "system scheduler" to refer to the scheduler that schedules jobs for your system, regardless whether it is an HPC cluster or datacenter.

Server

Term for a (typically) "headless" computer used in a data center and managed by a system administrator in an IT department. See Host.

Simultaneous Multi-Threading

Processor core's ability to execute (in hardware) instructions from multiple, independent, software execution threads and track their states simultaneously.

SMT

See Simultaneous Multi-Threading.

Task

A single unit of work (HTC job) defined by Nitro task definitions in a user task file (list of HTC jobs now referred to as tasks) that Nitro can schedule and launch for execution as a single OS process.

Task file

The file containing the list of tasks that Nitro should execute.

Task Launch Thread

A Nitro worker "software execution thread" capable of launching one Nitro "task". Unless modified by a Nitro worker command line option, the worker's quantity of task launch threads is identical to the quantity of host OS cores made available to the worker by the system scheduler.

Thread

In SMT or hyper-threading, refers to a hardware-based thread execution capability. For example, the consumer-oriented Intel Core i7 processor has four cores, each of which has hardware that can simultaneously track two software execution thread states and execute the other thread when one thread blocks waiting on a memory access; thus increasing the utilization of each core's computational capability and yielding 8 hardware-based threads for the entire processor. The server-oriented Intel Xeon processor documentation refers to threads as "logical processors". Cray documentation uses the term threads relative to the SMT capability of the Intel Xeon processors in its XC systems. Regardless of vendor terminology, one thread in the context of SMT refers to the hardware capability for tracking and executing one software execution thread. The term threads used in the context of a processor core refers to the quantity of software execution threads the core can simultaneously track; e.g., 2 threads per Intel Xeon E5-2650 v3 core and 20 threads per Intel Xeon E5-2650 v3 processor. BIOS settings

enable or disable the SMT capability of SMT-capable processors and therefore determine at boot time whether a processor has only one thread per core or multiple threads per core. See Core and OS Core.

Virtual Core

Term often used to refer to a hardware-based thread of a core that is not the first thread (thread 0) within a core. If a processor has SMT or hyper-threading enabled, "thread 0" represents the core and the other threads 1-N represent "virtual cores". The only way to execute using just a core that has SMT/hyper-threading enabled is to use only thread 0 of the core and expressly enforce the non-use of the core's other threads or "virtual cores" through a CPUset or control-group (cgroup).

Worker

Nitro component responsible for executing the user's workloads specified by the task definitions in the Task file.

Workload

Generic term used in this guide to refer to some amount of work to be done, typically by executing one or more software applications.

Chapter 4 Troubleshooting

This chapter provides troubleshooting information.

In this topic:

Related Topics

Sources of Troubleshooting Information

These are common sources of reference for troubleshooting:

- **Job Output Files** - Any errors that Nitro reports should be reported on stderr and will be captured to your job's output files (if running Nitro through a job scheduler).

- **Job and Task Log Files in the Job Directory** -

  - Nitro writes a job log indicating the startup parameters, input files, configuration, and the main worker events and statistics. You can review the job log to determine if any tasks failed, timed out, or were invalid (an error parsing the task line).

  - The task log contains a listing of all task results from the job and includes stdout and/or stderr output.

- **Nitro debug logs** - In the job directory you will also find a "logs" directory with worker and coordinator logs. The logs are named according to the role (worker or coordinator), host, job, and process ID so that logs being written to the same directory will not overwrite logs from another Nitro job or worker with the same process ID.

Related Topics

Troubleshooting Task Errors

The Nitro job log, and the stdout output from the Nitro coordinator, lists the number of tasks completed, tasks successfully completed (exit code of 0), failed tasks (exit code other than 0), tasks that timed out (exceeded the "maxtime" task option), invalid tasks (tasks that the coordinator could not parse without errors), and tasks with insufficient resources.

> ℹ If you encounter a problem you are unable to solve, forward the log files according to your company's escalation process.

In this topic:

Task Command Line Errors

If the command line that you use to specify the task's command line contains an error (for example, the path is incorrect or you are attempting to run a script with noexecute permissions set), then the shell will output an error message to stderr that will be captured and stored in the task log file.

For example, if the task's command line references a binary that doesn't exist, you would see the following error in the task log file.

```
Job ID Task ID Line # Task Name Status  Ret  Hostname Start time   Duration UserCPU
SysCPU VirtMem PhysMem Labels  Output
------ ------- ------ --------- ------- ---- -------- ------------ -------- ------- --
---- ------- ------- ------ ------
EX01   3       3        S07T2303  Failure 127  node02   07:58:07.868   0.005  0.000
0.000      0       0          bash: /opt/framemaker/bin/framegen: No such file or
directory
EX01   4       4        S07T2304  Failure 127  node02   07:58:07.872   0.007  0.000
0.000      0       0          bash: /opt/framemaker/bin/framegen: No such file or
directory
EX01   5       5        S07T2305  Failure 127  node02   07:58:07.878   0.003  0.000
0.000      0       0          bash: /opt/framemaker/bin/framegen: No such file or
directory
```

Failed Tasks

Failed tasks were tasks that the worker executed, but have failed because the command line was not valid, or the task ran and returned an exit code other than 0. To diagnose the error, examine the task log file located in the job directory. See Track Job Progress on page 15.

You can also use nitrostat to list failed tasks. See nitrostat on page 28. To list failed tasks using nitrostat use the following command line.

```
/opt/nitro/bin/nitrostat <job id> -f
```

The following is an example of information provided by nitrostat showing failed tasks.

```
Job ID Task ID Line # Task Name Status  Ret  Hostname Start time   Duration UserCPU
SysCPU VirtMem PhysMem Labels  Output
------ ------- ------ --------- ------- ---- -------- ------------ -------- ------- --
---- ------- ------- ------- ------
EX01   3       3         S07T2303  Failure 127  node02   07:58:07.868    0.005   0.000
0.000     0       0         bash: /opt/framemaker/bin/framegen: No such file or
directory
EX01   4       4         S07T2304  Failure 127  node02   07:58:07.872    0.007   0.000
0.000     0       0         bash: /opt/framemaker/bin/framegen: No such file or
directory
EX01   5       5         S07T2305  Failure 127  node02   07:58:07.878    0.003   0.000
0.000     0       0         bash: /opt/framemaker/bin/framegen: No such file or
directory
```

The line number of the failed task in the task file is listed so you can easily identify which lines in the task file generated errors. If you need to verify the path to the task file used by Nitro you will find it both in the job log file and in the stdout output from the coordinator, which may also be recorded in the output files provided by your scheduler.

```
Nitro Environment
-----------------
Job Id      : EX01
Job path    : /home/jdoe/jobs/EX01
Job log     : /home/jdoe/jobs/EX01/nitro_EX01.joblog.txt
Task log    : /home/jdoe/jobs/EX01/nitro_EX01.tasklog.txt
Task file   : /home/jdoe/jobs/example1.tasks
Worker hosts : node02
```

If Nitro cannot access the task file, you will receive an error from the coordinator on stderr indicating that the task file was not found or is not accessible.

Invalid Tasks

Invalid tasks are lines in the task file that Nitro could not parse without errors. Parsing errors usually include misspelling a task option name. If your task line doesn't contain any task options, you should prepend the "cmd=" option to your command line. The "cmd" option indicates that Nitro should stop parsing the task line and accept the rest of the line as the command line to be executed. The following example shows an invalid task.

```
Job ID Task ID Line # Task Name Status   Ret       Output
------ ------- ------ --------- -------- ---- ... ------
EX01   14      14              Invalid        Unrecognized option name: walltime
```

In this case an invalid option "walltime" was used in a task definition instead of "maxtime".

```
# invalid line:
name=S07T2314 walltime=30 cmd=/opt/framemaker/bin/framegen -i /shared/scene07.def -
tindex 2314

# valid line:
name=S07T2314 maxtime=30 cmd=/opt/framemaker/bin/framegen -i /shared/scene07.def -
tindex 2314
```

The task file may also be rejected if you have any binary data in the file. The task file should only include ASCII text and each task must be on a separate line. The task file allows comment and empty lines. A hash symbol (#) in the first column of a line identifies a comment line.

Insufficient Resources Tasks

If workers are not able to fulfill resource requirements for tasks with cores or memory specifications, an insufficient resources error is logged. The following example show a task with insufficient resources.

```
JobID   TaskID  Line Name        Status   ExitCode     Hostname      StartTime                    Du
UserCPU SystemCPU VirtualMem  PhysicalMem      Labels  Output
425     1       13               Success  0            node02        2016-02-16_17:58:07.052-
0700    0.030   0.010    0.000   130211840     1077248           1 1000000 3.142116000
425     2       14               Success  0            node02        2016-02-16_17:58:07.052-
0700    0.030   0.010    0.000   130211840     1081344           2 1000000 3.141296000
425     3       15               Success  0            node02        2016-02-16_17:58:07.083-
0700    0.021   0.010    0.000   275505152     1679360           3 1000000 3.139544000
425     4       16               InsufRes -1           node02        2016-02-16_17:58:07.083-
0700    0.000   0.000    0.000         0             0                          Error: worker
to 2 threads, task is requesting 3 threads
```

Related Topics