# FastDL

# TaskDL ®

Task Farming Framework for IDL

Version 2.4.0

Tech-X Corporation
5621 Arapahoe Avenue, Suite A
Boulder, CO 80303
http://www.txcorp.com
info@txcorp.com

# Contents

# 1 Preface

Names of functions, parameters, and other code samples are denoted in a fixed font — for example, `ls -la`.

Terms requiring a specific definition will appear in *italics* on first use.

TaskDL was built and tested on various Linux and Macintosh platforms. For support on other platforms (such as Solaris or Windows), please contact support@txcorp.com.

# 2 Introduction to TaskDL

## 2.1 Overview

Many experiments and simulations generate large data sets that must be processed quickly. Scientists exploring fluid and particle dynamics, high-energy and plasma physics, biophysics, protein folding and medical imaging are challenged visualizing and analyzing complex data. In response, many scientists and developers rely on the Interactive Data Language (IDL) from ITT Visual Information Solutions to visualize and analyze these large data sets.

However, some analysis cannot be practically accomplished on a workstation or server with only symmetric multi-processing and multi-threading due to size or time constraints. To overcome these barriers, networks of processors are used in a technique called *parallelization* or *parallel computing*. By distributing a large, complex problem over many CPUs, more computations can be performed in the same amount of time. Hardware that can provide parallel computing resources cover a wide range: this includes individual machines with SMP or multi-core CPUs, collections of workstations on a common network (Beowulf clusters), parallel "supercomputers" involving thousands of nodes and specialized networking interconnects, and geographically-distributed clusters (Grids).

Problems that can be solved using parallel computing fall into one of two catagories: *tightly-coupled* and *loosely-coupled* algorithms. These terms refer to how data or processing is divided among processors. tightly-coupled problems require a great deal of communication between processors. An example of this might be a weather simulation, where different geographical areas were processed by different CPUs. Since the temperature or air pressure of one area effects other locations, the processors would need to communicate frequently with each other in order to produce an accurate simulation.

In contrast, loosely-coupled problems require little or no communication between processors. A common example of this would be rendering images from a data set for later animation. Each image could be created without dependencies on any of the other images, every processor could be working on its own frame. These are sometimes referred to as "embarrassingly parallel" problems, as almost no effort is required to allow these problems to take advantage of parallel processing. For tightly-coupled problems, new algorithms must be used to take advantage of parallelization.

Using parallelization can result in dramatic, cost-effective increases in computing power; unfortunately, IDL does not naturally take advantage of a parallel environment. To bridge the gap between IDL and parallel computing, Tech-X Corporation has developed FastDL. With FastDL, scientists and developers can run IDL visualization and analysis applications in parallel on Linux clusters, significantly shortening the time required to get results.

FastDL is made up of two parts: mpiDL and TaskDL. The former was developed for tightly-coupled applications, calculations that require communication between processors. mpiDL allows for parallel algorithms to be implemented and used, and combines the power of the industry-standard Message Passing Interface (MPI) with the ease of use and varied visualization and analysis capabilities of IDL.

In contrast, TaskDL is a task-farming solution designed for problems with loosely-coupled applications where no communication between nodes of a cluster is required. This allows users to federate computers on an existing cluster or network, putting them to work on a common problem.

## 2.2 Architecture of TaskDL

The most basic component of TaskDL, as the name suggests, is the *task*. A task is simply a string containing a sequence of IDL commands; in other words, work that the user needs to complete. TaskDL uses a client-server model common in computer science; thus, tasks are collected and managed by the *TaskDL server (tdl)*. This servers is an application that is responsible for keeping track of tasks, their status, etc. The TaskDL server manages tasks using one or more *task queues* — ordered lists of tasks.

Of course, no goal can be achieved without effort; with TaskDL, this effort is provided by the *workers*. Workers are IDL sessions that is connected to (and communicating with) the tdl server. Workers wait for the tdl server to hand them a task, at which point they execute it and wait for the next one. Each worker has his own queue on the tdl server, called a *private queue* — this queue contains tasks designated for that specific worker. Workers receive tasks in their private queue first from the tdl server; if their private queue is empty, the server begins giving them tasks from the *default queue*, a queue for tasks that are not specific to a given worker

Note that tasks are executed in a persistent session, so results of one task are available in the next task. If a worker dies unexpectedly or is otherwise forced to quit, the tasks in its private queue are moved to the default queue.

## 2.3 Communication within TaskDL

The TaskDL software does not need to be installed on each worker node in order to federate them into a task farm. However, each distributed worker must have an IDL runtime or full license.

Worker nodes are launched from a master node using SSH or RSH. If SSH is used, the cluster environment should be set up to allow public key authentication of `ssh` so that passwords do not need to be entered for every host which is added to the task farm. If `rsh` is used, then the appropriate `.rhosts` and other configuration files should be installed. Consult your system administrator for more information on public key authentication of `ssh` and the use of `rsh`.

## 2.4 TaskDL workflow

A typical workflow for a TaskDL session is as follows:

1. IDL is launched on one of the nodes. This node is often referred to as the *master node*.
2. The master node instantiates a TaskDL object using `OBJ_NEW('TaskDL')` and opens a session with `TaskDL::open_session`.
3. The master node then commissions one or more remote worker nodes using the `TaskDL::add_worker` method. `ssh` (or `rsh`) is used to log on to a remote host, set needed environment variables, and then to start a TaskDL IDL process on the worker.
4. Tasks may be added to the task queue through the `TaskDL::add_task` method. Tasks are simply IDL routines to be run on the worker nodes, along with any input arguments or keywords. The task may be an IDL routine written by the user or a system routine.
5. The tdl server distributes tasks to workers. As workers complete tasks, they receive new ones.
6. Once all tasks are complete, the user can either keep the session active for future tasks, or shutdown the TaskDL instance.

## 3 TaskDL Installation

Installation of TaskDL is straightforward.

1. Expand the package with

   ```
   tar -xvzf TaskDL-2.4.0.tar.gz
   ```

2. The resulting directory, `taskdl-2.4.0`, can now be copied to your chosen installation location, such as `/opt`, with a command similar to

   ```
   cp -R taskdl-2.4.0 /opt
   ```

   Note that you may need root access to install TaskDL in certain locations, such as `/opt`.

After the files are copied, there are several environment variables that need to be set. Rather than setting them in your current shell, you should modify your shell's startup file (e.g., .bashrc for bash, and .cshrc for csh) to include the following lines, because these variables will be needed each time a worker is spawned.

- `TASKDL2_DIR` needs to point to the top level of the TaskDL installation; for example, `/opt/taskdl2/`
- `TASKDL2_DIR/idl` needs to be included in your `IDL_PATH`; for example, for bash or other sh shells:

  ```
  export IDL_PATH="$TASKDL2_DIR/idl:<IDL_DEFAULT>"
  ```

  or

  ```
  export IDL_PATH="$TASKDL2_DIR/idl:$IDL_PATH"
  ```

  if you already have previously set IDL_PATH.

  For csh-based shells:

  ```
  setenv IDL_PATH "$TASKDL2_DIR/idl:<IDL_DEFAULT>"
  ```

  or

  ```
  setenv IDL_PATH "$TASKDL2_DIR/idl:$IDL_PATH"
  ```

  if you already have previously set IDL_PATH.
- `TASKDL2_SESSIONROOT` is optional; you can set it to point to the root directory of where TaskDL should create session directories. If this variable is not specified, the remote worker will attempt to write any files to the working directory of the interactive session. If this path does not exist, it defaults to `$HOME` as the session root.

# 4 Basic Usage of TaskDL

This section will demonstrate basic usage of TaskDL — starting a server, creating tasks, commissioning workers, and reviewing results. For more advanced use (including multiple queues, prioritizing tasks, or task groups), see Section 5.

### 4.0.1 Creating a TaskDL Object

TaskDL uses IDL's object interface; as a result, an object needs to be instantiated for TaskDL. This can be done with the following command:

```
IDL> oFarm = OBJ_NEW('taskdl')
```

### 4.0.2 Opening a Session

Once a `taskdl` object has been created, a new session needs to be opened. This will start the tdl server and create the default task queue. To open a new session, use the following command:

```
oFarm->open_session
```

### 4.0.3 Spawning a Worker

In order to process tasks, one or more workers need to be commissioned. A worker process can exist on the local machine or a remote host; if the worker is not local, TaskDL attempts to use SSH (by default) to make the remote connection.

To spawn a new worker, issue the following command:

```
 oFarm->spawn_worker
```

By default, this method creates a new worker on the local machine, i.e. `localhost`. This default is not only for convenience; as more and more PCs have SMP or multi-core processors, spawning multiple workers on a local machine can use resources more efficiently.

Workers also have a *session directory* that is used to store any data that may be written. By default, this will be the working directory of the IDL session that spawned the worker. If the worker is on a machine that is unable to write to that file system, it attempts to create session local to the worker's machine. If this also fails, the worker then looks in the `TASKDL2_SESSIONROOT` directory. Also, note that session directory names are usually relative to `TAKSKDL2_SESSIONROOT`; that is, sessions will be (by default) placed in

```
$TASKDL2_SESSIONROOT/taskdl_session
```

if the `TAKSKDL2_SESSIONROOT` environment variable is defined. This path can be specified in the `open_session` method.

Finally, note that `stdout` is not returned to your host IDL session. Output that would normally be written to the screen is instead logged in your session directory.

### 4.0.4 Adding a Task

Tasks are strings of one or more valid IDL commands. This may be something as simple as printing “Hello, world” to restoring a `.sav` file, running several procedures, and writing out images of the results. Simple or complex, the method of adding a task to the default queue is the same:

```
oFarm->add_task, 'print, "Hello Taskfarm"'
```

### 4.0.5 Ending a Session

To close a session and terminate the tdl server, use the following command:

```
oFarm->close_session
```

Note that this will wait until all tasks in all queues are completed before terminating the tdl server.

### 4.0.6 Exiting TaskDL

When TaskDL is no longer needed, the object can be destroyed. This is done using the same commands as cleaning up any other IDL object:

```
obj_destroy, oFarm
```

# 5 Advanced Usage of TaskDL

This section will demonstrate more advanced usage of TaskDL, including private queues, prioritizing tasks using stages, and task groups. If you are not sure how to start up TaskDL, create tasks, or commission workers, please see Section 4.

### 5.0.7 Private Queues

When a worker is spawned, by default it will receive tasks from the (inventively named) `default` queue. You can also create a private queue for that worker with the following command:

```
oFarm->spawn_worker, QUEUEID=queueid
```

To then add a task to that private queue, use the following:

```
oFarm->add_task, 'print, "hello from a specific worker"', QUEUEID=queueid
```

TaskDL workers will process tasks in their private queue first; if their private queue is empty, they will begin completing tasks from the default queue.

## 5.1 Task Stages

Some problems have dependencies between various tasks – some work can be done concurrently, while others must wait for previous work to finish. A common example of this is generating an animation; frames can be rendered from data in parallel, but combining frames into a single file must only be done after all rendering is complete. This problem can be solved in several ways in TaskDL; one method is using *task stages*.

Task stages allow the user a way to show dependencies between tasks, as well as showing which tasks can or cannot be done concurrently. TaskDL will only start processing tasks at a higher stage once all tasks at a lower stage are completed. Stages are represented by an integer stored in the `STAGE` keyword for various methods.

For example, in generating an animation using stages, a user might first do the following:

```
for i = 0, nframes-1 do $
  oFarm->add_task, 'render_image,'+string(i), STAGE = 1

oFarm->add_task, 'encode_movie', stage=2

oFarm->advance_stage, 2
```

The `advance_stage` method instructs the TaskDL server to increase stages one by one until all tasks up to and including the level specified have been completed. In this example, TaskDL will begin processing all stage 1 tasks. Once those have finished, the server will have workers process stage 2 tasks. When stage 2 has finished, the server will advance to stage 3.

To set (or reset) the server's current stage manually, use the `set_stage` method:

```
oFarm->set_stage, 1
```

## 5.2 Task Groups

*Task groups* give the user another, more flexible way to organize tasks and the order in which they are processed. This feature allows tasks to be collected in some logical fashion, and then executed based on relationships with other groups. Tasks in a group are done concurrently, and can have dependencies on other groups. Returning to our movie generation example, rendering frames might be one group, while processing those frames into a movie might be a second group. Finally, TaskDL can be told that group two depends on group one, ensuring that all tasks were processed in the correct order. Implementing movie generation using groups in TaskDL might look like the following:

```
oFarm->create_group, GROUP='render'
oFarm->create_group, GROUP='create_movie'
oFarm->group_dependency, GROUPHIGH='render', GROUPLOW='create_movie'

for i = 0, nframes-1 do oFarm->add_task, 'render_image,'+string(i), GROUP='render'

oFarm->add_task, 'encode_movie', GROUP='create_movie'
```

Note that the `group_dependency` method *does* allow for circular dependencies! If a user tells TaskDL that group A depends on group B and group B depends on group A, the results are undetermined — the user should take caution to ensure that these errors are not created when writing code for TaskDL.

# 6 Examples

## 6.1 Example 1: `hello_taskdl.pro`

The following example shows a basic use case for TaskDL — all of the steps necessary for starting TaskDL, spawning workers, submitting tasks, and cleaning up the session afterwords are shown.

```
PRO hello_taskdl
; create a tasdl object, which will be our main interface to taskdl
 oFarm = OBJ_NEW('taskdl')

; create a new taskfarming session. This launches the tdl server,
; creates the session directory and creates the default queue.
 oFarm->open_session

; spawn a worker which will connect to the tdl server
 oFarm->spawn_worker

; add a task to the default queue
 oFarm->add_task, 'print, "Hello Taskfarm"'

; Close the session, terminate the tdl server if all tasks are
; completed
 oFarm->close_session

; cleaup
 obj_destroy, oFarm
END
```

### 6.2 Example 2: `hello_groups.pro`

This example demonstrates using groups in TaskDL to represent dependencies between tasks.

```
PRO hello_groups
; create a tasdl object, which will be our main interface to taskdl
 oFarm = OBJ_NEW('taskdl')

; create a new taskfarming session. This launches the tdl server,
; creates the session directory and creates the default queue.
 oFarm->open_session

; create some task groups
 oFarm->create_group, 'A'
 oFarm->create_group, 'B'
 oFarm->create_group, 'C'

; group B depends on A
; no tasks in group will be released until completion of all
; tasks in group A
 oFarm->group_dependency, 'B', 'A'
; group C depends on B
 oFarm->group_dependency, 'C', 'B'

; add some tasks to the default queue
 oFarm->add_task, 'print, "Hello from group C"', group = 'C'
 oFarm->add_task, 'print, "Hello from group B"', group = 'B'
 oFarm->add_task, 'print, "Hello from group A"', group = 'A'

; spawn a worker which will connect to the tdl server
 oFarm->spawn_worker

; Close the session, terminate the tdl server if all tasks are
; completed
 oFarm->close_session

; cleanup
 obj_destroy, oFarm
END
```

### 6.3 Example 3: `hello_stages.pro`

This example demonstrates use of stages in TaskDL.

```
PRO hello_stages

; create a tasdl object, which will be our main interface to taskdl
 oFarm = OBJ_NEW('taskdl')

; create a new taskfarming session. This launches the tdl server,
; creates the session directory and creates the default queue.
 oFarm->open_session, host=localhost, session_dir = './hello_stages'

; spawn a worker which will connect to the tdl server
; Each worker has a queue associated with it and we keep these
; queue IDs for later identification of the workers.
 oFarm->spawn_worker, queueID = queue1
 oFarm->spawn_worker, queueID = queue2

; add a task to each of the workers. These tasks contain the
; initialization of a on both workers.
 oFarm->add_task, 'print, "Hello worker 1" & a = 0', queueId=queue1
 oFarm->add_task, 'print, "Hello worker 2" & a = 0', queueId=queue2

; add some tasks to the default queue. The workers will compete for
; these tasks.
;
; We don't run into danger that variable 'a' is not initialized, as the
; the following tasks are on the same stage as the initialization tasks.
; Within a stage, tasks are executed in the order they were added.

 for i = 0, 10 do $
     oFarm->add_task, 'a = a + 1'

; both workers report their result:
; The value reportd for a will depend on the exact timing of the two
; workers.  One worker will report A as the current value of a, the
; other worker will report 11 - A.
;
; As these tasks are on a higher/later stage than the previous tasks.
; Therefore we know that none of the above increment tasks will still
; be available once the following tasks start.

 oFarm->add_task, 'print, "Worker 1 summation result: a = ", a', $
              queueId=queue1, stage = 1
 oFarm->add_task, 'print, "Worker 2 summation result: a = ", a', $
              queueId=queue2, stage = 1

; worker 1 sends its result to worker 2:
;
; To accomplish this, we insert a task for worker 1 which inserts a
; task for worker 2. The advantage of this is that at the time we insert
; this task, we don't need to know the value of 'a', as this task will
; be created at runtime on worker 1.
```

```
 oFarm->add_task, $
    'self->add_task, "b = a + "+string(a), queueId='+strtrim(queue2, 2)+$
                 ', stage=1', queueId=queue1, stage = 1

; and now the same on worker 2:
 oFarm->add_task, $
    'self->add_task, "b = a + "+string(a), queueId='+strtrim(queue1, 2)+$
                 ', stage=1', queueId=queue2, stage = 1

; both workers report their result.
; This has to happen after all the previous tasks have been completed.
; Therefore they are added on the next higher stage.
 oFarm->add_task, 'print, "after communication, b = ", b', $
                 queueId=queue1, stage = 2
 oFarm->add_task, 'print, "after communication, b = ", b', $
                 queueId=queue2, stage = 2

; up to this point, the tasks were added to the tdl server, but
; only tasks at stage 0 were released to the workers. We advance the
; stage now to 2, which will make all tasks available to the workers.
 oFarm->advance_stage, stage=2

; we now close the session
 oFarm->close_session

; cleanup
 obj_destroy, oFarm

END
```

# 7 API Reference

## 7.1 TaskDL Methods

### 7.1.1 `connect`

This function connects to an existing TaskDL server.

If the connection is successful, this function sets data members host, port and unit for the TaskDL object; otherwise, it sets an error code.

#### Syntax

myTaskDL->connect [, HOST=host] [, PORT=port]

#### Return value

None.

#### Arguments

None.

#### Keywords

**HOST**

A scalar string that denotes the name of the host on which the TaskDL server is running. Default: localhost.

**PORT**

An integer denoting the port to use when connecting to the TaskDL server. Default: 40001

### 7.1.2 `disconnect`

This function disconnects from the current TaskDL server. If no connection to a server currently exists, this method does nothing.

#### Syntax

myTaskDL->disconnect

#### Return value

None.

#### Arguments

None.

#### Keywords

None.

### 7.1.3 `create_session`

This function creates a new TaskDL session. If a TaskDL server is currently running on the specified machine, this method will connect to it. If no TaskDL server can be found, one will be created.

## Syntax

myTaskDL->create_session [, HOST=host] [, PORT=port]

## Return value

None.

## Arguments

None.

## Keywords

**HOST**

A scalar string that denotes the name of the host on which the server is running.

**PORT**

An integer denoting the port to use when connecting to the TaskDL server.

### 7.1.4 `spawn_worker`

This function creates a new TaskDL worker in the current session.

## Syntax

myTaskDL->spawn_worker [, HOST=host] [, /RSH] [,IDLCMD=command] [,/full] [, /TUNNEL] [, /QUEUEID]

## Return value

None.

## Arguments

None.

## Keywords

**HOST**

The name of the host that is to be enlisted as a worker. Default: localhost.

**RSH**

Indicates that RSH should be used instead of SSH when connecting to the worker's host.

**IDLCMD**

A string stating the command to be used as the IDL executable. This is useful if IDL is launched through a script that sets up the environment.

**full**

If set, this keyword will cause the worker to use a full IDL license on the remote host. Default: off (and therefore, use an IDL runtime license)

**TUNNEL**

If set, this keyword causes the method to use ssh tunneling for port forwarding. This is useful if the worker sits behind a firewall.

**QUEUEID**

The ID of the newly created queue associated with the worker. This is the queue for which the new worker will be responsible, i.e. its private queue.

### 7.1.5 `kill_worker`

This command kills a TaskDL worker with an interrupt. Any current tasks assigned to the worker are lost.

## Syntax

myTaskDL->kill_worker[, QUEUEID = queueid]

## Return value

None.

## Arguments

None.

## Keywords

**QUEUEID**

The ID associated with the worker to kill.

### 7.1.6 `create_queue`

This method instructs the current TaskDL server to create a new queue, to which workers can then connect. Once created, workers can now add tasks to this queue ID (even though there might not yet be a worker connected to the queue).

## Syntax

queueID = myTaskDL->create_queue()

## Return value

Returns the ID of the newly created queue.

## Arguments

None.

## Keywords

**WORKER_HOST**
The name of the host that is to be enlisted as a worker.

### 7.1.7 `create_group`

This method creates a task group in the current session.

## Syntax

myTaskDL->create_group [, GROUP=group]

## Return value

None

## Arguments

None.

## Keywords

**GROUP**

An arbitrary string identifying the group.

### 7.1.8 `group_dependency`

This method is used to describe a dependency between task groups.

Note: this method does allow creation of circular dependancies, i.e. group A depending on group B while group B depends on group A. It is the responsibility of the user to ensure that these circular dependencies are not caused.

## Syntax

myTaskDL->group_dependency [, GROUPHIGH=grouphigh, GROUPLOW=grouplow]

## Return value

None

## Arguments

None.

## Keywords

**GROUPHIGH**

A string identifying the higher-priority task group. Tasks in this group are completed first.

**GROUPLOW**

A string identifying the lower-priority task group. Tasks in this group are not started until the tasks in GROUPHIGH have completed.

### 7.1.9 `remove_queue`

This method removes a queue from the TaskDL server. Any tasks left in this queue are transferred to the server's default queue.

## Syntax

myTaskDL->remove_queue, QUEUEID=queueID

## Return value

None.

## Arguments

None.

## Keywords

**QUEUEID**

The string identifying the queue to be deleted.

### 7.1.10 `connect_queue`

This method connects to a queue that already exists on the TaskDL server, and is called once a worker starts up. The worker gets the queue ID as a command line argument and connects under this ID to the TaskDL server.

#### Syntax

myTaskDL->connect_queue, QUEUEID=queueID

#### Return value

None.

#### Arguments

None.

#### Keywords

**QUEUEID**
The ID identifying the queue to be connected.

### 7.1.11 `get_queueID`

This method returns the ID of the queue associated with the current object.

#### Syntax

queueID= myTaskDL->get_queueID()

#### Return value

queueID: an integer handle to the requested queue.

#### Arguments

None.

#### Keywords

None.

### 7.1.12 `show_queues`

This method is used to display the queues of the session.

## Syntax

myTaskDL->show_queues [, QUEUES=queues]

## Return value

None.

## Arguments

None.

## Keywords

**QUEUES**

If this keyword is specified, output is stored in this variable instead of being output to the display.

### 7.1.13 `get_queues`

This method returns an array of the currently active queue IDs.

## Syntax

queue_list = myTaskDL->get_queues()

## Return value

An array containing all queue IDs for the TaskDL object. If an error occurs, the method will return 0.

## Arguments

None.

## Keywords

None.

### 7.1.14 `list_queue`

This method lists all tasks in a given queue.

## Syntax

myTaskDL->list_queue [, QUEUEID] [, TASKS=tasks]

## Return value

None.

## Arguments

None.

## Keywords

**QUEUEID**
The ID of the queue to list.

**TASKS**
If specified, this variable will contain a list of tasks in the requested queue.

### 7.1.15 `reset_server`

This method removes all tasks from all queues on the TaskDL server. This creates a "clean state" to start an entirely new workflow or to recover from errors.

## Syntax

myTaskDL->reset_server

## Return value

None.

## Arguments

None.

## Keywords

None.

### 7.1.16 `close_session`

This method closes a TaskDL session.

## Syntax

myTaskDL->close_session

## Return value

None.

## Arguments

None.

## Keywords

None.

### 7.1.17 `set_stage`

Release all tasks on a lower or equal stage. The tasks will still be processed according to their stages.

## Syntax

myTaskDL->set_stage [, STAGE=stage]

## Return value

None.

## Arguments

None.

## Keywords

**STAGE**
The stage up to which tasks will be released.

### 7.1.18 `advance_stage`

This method advances the TaskDL object to the specified stage.

## Syntax

myTaskDL->set_stage [, STAGE=stage]

## Return value

None.

## Arguments

None.

## Keywords

**STAGE**
The stage to which the server will be advanced.

### 7.1.19 `add_task_to_all`

This method adds a given task to all queues. This is useful when assigning actions to all workers, such as gathering a result or writing out final data.

## Syntax

myTaskDL->add_task_to_all, TASK [, STAGE=stage]

## Return value

None.

## Arguments

None.

## Keywords

**TASK**

The task to be added to all queues.

.

**STAGE**

The priority given to the added task.

### 7.1.20 `add_task`

This method adds a given task to a given queue.

## Syntax

myTaskDL->add_task, task [, QUEUEID=queueid] [, STAGE=stage] [, GROUP=group]

## Return value

None.

## Arguments

None.

## Keywords

**task**

The task to be added to all queues.

**QUEUEID**

The queue to which the task should be added. Default: 0

**STAGE**

The stage given to the added task. Default: 0

**GROUP**

The group to which the task is assigned. Default: default

### 7.1.21 `remove_task`

This command removes a task from a given queue.

## Syntax

myTaskDL->remove_task, task [, QUEUEID=queueid] [, TASKID=taskid]

## Return value

None.

## Arguments

None.

## Keywords

**QUEUEID**

The queue from which the task should be deleted.

**TASKID**

The task number to delete.

This method adds a given task to a given queue.

### 7.1.22 `get_task`

This method returns the next available task. retrieves the next available task from the TaskDL server. It returns tasks by looking in the following order:

1. the private queue
2. the default queue

Within a queue, tasks are handed out according to their priorities.

## Syntax

task = myTaskDL->get_task()

## Return value

None.

## Arguments

None.

## Keywords

None.

### 7.1.23 `get_task_for_self`

This method returns the next available task from the private queue; it does not check the default queue for tasks. Within a queue, tasks are handed out according to their priorities.

#### Syntax

task = myTaskDL->get_task_for_self()

#### Return value

None.

#### Arguments

None.

#### Keywords

None.

### 7.1.24 `finish_task`

This method informs the TaskDL server about the exit status of a task. If a task has failed during execution, TaskDL will not attempt to execute the task again.

## Syntax

myTaskDL->finish_task, STATUS = status

## Return value

None.

## Arguments

None.

## Keywords

**STATUS**

The IDL return code of a task, with 1 indicating a success and 0 a failure.

### 7.1.25 `spin`

The spin procedure puts an IDL session into an indefinite loop in which the process will continuously poll the server for tasks with get_task() and execute any tasks received with the IDL execute function. This loop continues until a 'QUIT' string is received as a task.

## Syntax

myTaskDL->spin [, EXCLUSIVE = exclusive]

## Return value

None.

## Arguments

None.

## Keywords

**EXCLUSIVE**
Only check the private queue for tasks, rather than checking the private and default queue.

# 8 Troubleshooting Guide

## 8.1 "port bind" error

"I receive a "port bind" error when attempting to create a session"

The default port for TaskDL (40001) may be in use or otherwise blocked. To switch to a different port, use the command

```
myTaskDL->create_session, PORT=port
```

where `port` is the port number you wish to use. This number should be above 1024; ports below this are well-known services (such as ssh, ftp, and ntp) and will not be available.

## 8.2 Password prompts or timeouts using ssh

"I am trying to start a worker on a remote machine using ssh, but I get a password prompt or the request times out."

The best method of managing logins using scripted applications such as TaskDL is through public/private keys. Details on setting up key sharing between Unix machines can be found at

http://nosheep.net/story/password-less-ssh-login/

## 8.3 Using ssh on Windows

"I am using Microsoft Windows; how can I use TaskDL with remote machines?"

TaskDL relies on ssh (or rsh) for communication with remote machines, such as worker nodes. While almost all Unix machines (including Mac OS X) ship with ssh by default, computer using Windows often do not have ssh. However, there are several high-quality ssh client/server applications that are free for the Windows platform, including:

- PuTTY — http://www.chiark.greenend.org.uk/ sgtatham/putty/
- Cygwin — http://cygwin.com/

## 8.4 IDL license issues

"IDL is complaining about license issues in my interactive session or workers."

Workers in TaskDL can be spawned using either a full or runtime license; the latter can run IDL `.sav` files, but has the restriction that it cannot compile new code. If your workers have a runtime license, there are some additional step that you can take to have arbitrary IDL code in your tasks:

1. First, try spawning your workers with full licenses using `oFarm->spawn_worker, /full`.
2. If that fails, we must use runtime licenses. Write whatever IDL code you wish to be in a task, and compile this into a save file (see IDL documentation for details). Make sure that *all* routines are resolved, using the IDL command `resolve_all`.
3. Once you have an IDL save file, you can have a worker restore and run this file with the following:

```
    oFarm->spawn_worker, queueID=workerQueue
    oFarm->add_task, 'restore, "myProcedure.sav"', queueID=workerQueue
```

For more details, see the `demo_save_file.pro` example.

## 8.5 Error handling

"I think TaskDL is encountering an error; how do I check?"

All taskDL methods set a message field to announce status. This is stored in the data member `taskdl::tdl_msg`. There also exists a data member for error reports, `taskdl::tdl_error`.

## 8.6 Using scripts to launch IDL

"My organization uses a script to launch IDL. How do I use this script when spawning new workers?"

The `spawn_worker` method has a keyword that allows you to specify the command used to start new workers, such as `sswidl` on some systems:

```
oFarm->spawn_worker, cmd='sswidl'
```

# 9 Starting TaskDL Manually

TaskDL can spawn workers remotely on Unix systems. However, if your server or workers are running on Windows, workers currently need to be manually launched and configured. This information is also useful for non-Windows users. Both IDL and TaskDL are highly system- and installation-dependant in terms of where to find applications (ssh), IDL, and needed environment information that the automatic spawning of workers may not work on all systems.

## 9.1 Starting the Server

The tdl server can be started manually from the command line if needed. To do so, simply run the following:

```
tdl <port number>
```

where <port number> is the port tdl should use to communicate.

## 9.2 Spawning a Worker

The following steps are performed by TaskDL when spawning a worker — i.e. launching a new worker on a remote machine. If you are using Windows workers, these steps will have to be performed manually.

Regardless of platform, the installation directory of TaskDL should be included in the `IDL_PATH` environment variable.

1. A connection to the remote machine is made using ssh. taskDL then launches IDL in the correct license mode.

   After an IDL session is started, a new instance of TaskDL is created using the command

   ```
   worker = obj_new('taskdl')
   ```

2. After the TaskDL object is created, it is connected to the server using the command

   ```
   worker->connect, HOST=host, PORT=port
   ```

   The values for `host` and `port` will correspond to the location and port number of the target server. (The default host is localhost, and the default port used by TaskDL is 40001.)

3. After connecting to the server, the worker is connected a queue with

   ```
   worker->connect_queue
   ```

4. The final step is to have the worker begin polling the server for new tasks to execute in the queue. This is done with the command

   ```
   worker->spin()
   ```

If you create a worker in the Linux environment via `spawn_worker` method, TaskDL initially informs the tdl serer that a worker is going to connect and allocates a queue for them. It will then ssh over and execute the above tasks, connecting that worker to the newly-created queue. If workers are started by hand, however, you may not know if there is an existing queue for that worker. In this case, a queue ID is not specified — this causes the tdl server to assign a queue for the worker automatically.

# 10 Additional APIs

In addition to using IDL for workers, one can connect to and employ TaskDL with other languages, including Python, Java, and C++. TaskDL workers written in any of these languages can connect to the same tdl server. This allows users both flexibility when choosing what language to use when solving a given problem and allows heterogenous task farms.

## 10.1 Python API

The Python API can be used by importing the taskPython module into your Python script:

```
from taskPython import *
```

One difference from the IDL API is in creation of a TaskDL object; in Python, this is done using the following function.

```
myTaskDL = TxTaskFarmInterface()
```

The following are examples of how the Python API is used. For complete details of the functions, see the IDL documentation in Section 7.

```
myTaskDL.connect(host = 'localhost', port = 40001, status = 0)

myTaskDL.disconnect()

myTaskDL.createSession(host = 'localhost', port = 40001, dir = os.getcwd())

workerID = myTaskDL.spawnWorker(workerHost = 'host')

queueID = myTaskDL.createQueue()

myTaskDL.createGroup(groupLabel)

myTaskDL.groupDependency(groupHigh, groupLow)

myTaskDL.showGroups()

myTaskDL.removeQueue(queueID, stage = 0)

myTaskDL.connectToQueue(queueID)

queueID = myTaskDL.getQueueID()

myTaskDL.showQueues()

queueArray = myTaskDL.getQueues()

myTaskDL.listQueue(queueID)

myTaskDL.resetServer()
myTaskDL.closeSession()

myTaskDL.killServer()

myTaskDL.initiateShutDown(stage = 0)

myTaskDL.setStage(stage = 0)

myTaskDL.advanceStage(stage = 0)

myTaskDL.addTaskToAll(task, stage = 0)

myTaskDL.addTask(task, queueID = 0, stage = 0, group = 'default')

myTaskDL.addTasksFromFile(task, queueID = 0, stage = 0)

task = myTaskDL.getTask()

task = myTaskDL.getTaskForSelf()

myTaskDL.finishTask(status)

myTaskDL.spin()
```

## 10.2 C++ API

TaskDL also provides an interface so that workers can be written in C++. This interface is defined in the file `TxTaskFarmInterface.h`, and should be included in C++ applications that wish to interact with a TaskDL server. The signatures for functions defined in this API are shown below. For complete details of the functions, see the IDL documentation in Section 7.

```
class TxTaskFarmInterface {

public:

  TxTaskFarmInterface(void);

  void    connectToServer(string host, int port);
  void    disconnectFromServer(void);

  int     createQueue(void);
  void    createGroup(string groupLabel);
  void    groupDependency(string groupHigh, string groupLow);
  void    showGroups(void);
  void    removeQueue(int queueID, int priority);
  void    connectToQueue(int queueID);
  int     getQueueID(void);
  void    showQueues(void);
  void    listQueue(int queueID);
  void    resetServer(void);
  void    closeSession(void);
  void    initiateShutDown(int priority);
  void    setStage(int priority);
  void    advanceStage(int priority);

  void    addTaskToAll(string task, int priority);
  void    addTask(string task, int queueID,
                  int priority, string groupLabel);
  string  getTask(void);
  string  getTaskForSelf(void);
  void    finishTask(int status);
};
```

## 10.3 C++ Example

```
#include <iostream>
#include "TxTaskFarmInterface.h"
using namespace std;

int main(int argc, char *argv[]) {

  if(argc != 3) {
    cerr << "usage: " << argv[0] << " host port " << endl;
    exit(1);
  }

  string host = argv[1];
  int port = atoi( argv[2] );

  TxTaskFarmInterface myTaskDL;

  myTaskDL.connectToServer(host, port);

  int queueID = myTaskDL.createQueue();

  myTaskDL.connectToQueue(queueID);

  myTaskDL.showQueues();

  myTaskDL.initiateShutDown(3);

  int nTasks = 10;

  for (int k = 0; k < nTasks; k++) {
    myTaskDL.addTask("test", 0, 1, "default");
  }

  myTaskDL.listQueue(0);

  myTaskDL.advanceStage(1);

  for (int k = 0; k < nTasks; k++) {
    cout << myTaskDL.getTask() << endl;
    myTaskDL.finishTask(0);
  }

  myTaskDL.listQueue(queueID);

  myTaskDL.advanceStage(3);
}
```

## 10.4 Java API

TaskDL also provides an interface so that workers can be written in Java This interface is defined in the file `TxTaskFarmInterface.java`, and should be included in Java applications that wish to interact with a TaskDL server. The signatures for functions defined in this API are shown below. For complete details of the functions, see the IDL documentation in Section 7.

```
  public class TxTaskFarmInterface {

    static void connectToServer(String hostName, int port);
    static void disconnectFromServer();

    static int createQueue();
    static void createGroup(String groupLabel);
    static void groupDependency(String groupHigh, String groupLow);
    static void showGroups();
    static void removeQueue(int queueID, int stage);
    static void connectToQueue(int queueID);
    static void showQueues();
    static void listQueue(int queueID);

    static void resetServer();
    static void closeSession();
    static void initiateShutDown(int stage);
    static void setStage(int stage);
    static void advanceStage(int stage) ;

    static void addTaskToAll(String task, int stage);
    static void addTask(String task, int queueID, int stage, String group);
    static String getTask();
    static String getTaskForSelf();
    static void finishTask(int status);

} // TxTaskFarmInterface
```

## 10.5 Java Example

```
  public class TdlClient {

    public static void main(String[] args) throws Exception {

      if (args.length != 2) {
        System.err.println("Usage: java TdlClient host port");
        System.exit(1);
      }

      TxTaskFarmInterface
        myTaskDL = new TxTaskFarmInterface(args[0], Integer.parseInt(args[1]));

      int queueID = myTaskDL.createQueue();

      myTaskDL.connectToQueue(queueID);

      myTaskDL.showQueues();

      myTaskDL.initiateShutDown(3);

      int nTasks = 10;

      for (int k = 0; k < nTasks; k++) {
        myTaskDL.addTask("test", 0, 1, "default");
      }

      myTaskDL.listQueue(0);

      myTaskDL.advanceStage(1);

      for (int k = 0; k < nTasks; k++) {
        System.out.println(myTaskDL.getTask());
        myTaskDL.finishTask(0);
      }

      myTaskDL.listQueue(queueID);

      myTaskDL.advanceStage(3);

      myTaskDL.disconnectFromServer();

    }
  } // TdlClient
```