Tigres Documentation

Release 0.2.0

Data Science and Technology, LBNL

July 08, 2016

CONTENTS

CHAPTER

ONE

WHAT IS TIGRES?

Template Interfaces for Agile Parallel Data-Intensive Science

Tigres provides a programming library to compose and execute large-scale data-intensive scientific workflows from desktops to supercomputers. DOE User Facilities and large science collaborations are increasingly generating large enough data sets that it is no longer practical to download them to a desktop to operate on them. They are instead stored at centralized compute and storage resources such as high performance computing (HPC) centers. Analysis of this data requires an ability to run on these facilities, but with current technologies, scaling an analysis to an HPC center and to a large data set is difficult even for experts. Tigres is addressing the challenge of enabling collaborative analysis of DOE Science data through a new concept of reusable **"templates"** that enable scientists to easily compose, run and manage collaborative computational tasks. These templates define common computation patterns used in analyzing a data set.

Tigres is inspired by the success of the **MapReduce** model. When the MapReduce model emerged from the Internet search space, it provided a radically simpler paradigm for parallel analysis by certain classes of applications. The simplicity of the API and analysis model enabled many applications to quickly script powerful scalable analyses. Similarly, Tigres provides abstractions that support a wide-array of common scientific application computational patterns. The goal of Tigres is to:

1. Provide template abstraction to capture the core set of fundamental workflow patterns, which will allow users to compose collaborative workflow scripts in a programming language of their choice.
2. Provide a hybrid execution mechanism for the templates that enables users to prototype their analysis workflows on desktops and seamlessly adapt them to run in production environments at scale.
3. Provide programmatic interfaces that will allow automated and user-provided provenance tracking.
4. Provide interfaces to capture execution state and allow users to understand complex parallel faults encountered during execution of the workflow.

CHAPTER

TWO

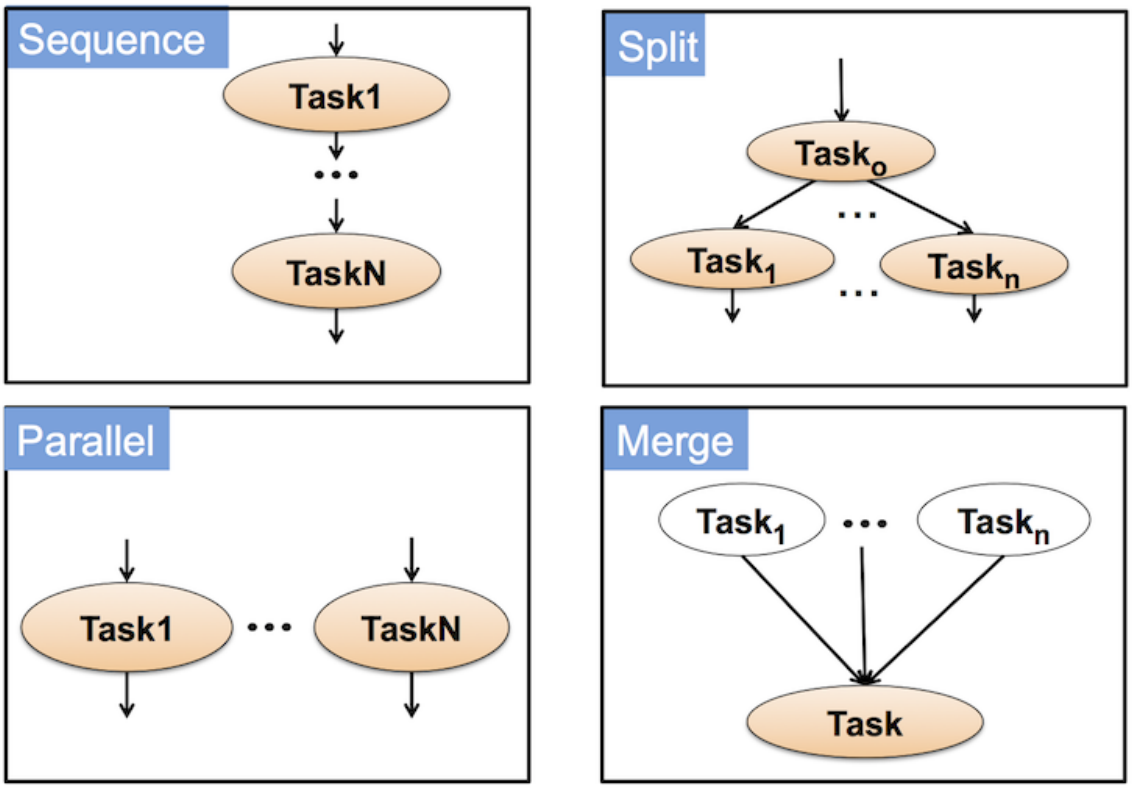
CONCEPTS IN TIGRES

Tigres provides an API for composing, executing and monitoring workflows using an abstraction called *templates* which captures common execution patterns. In this section of the documentation, we provide some overview information on the following topics:

2.1 Templates

The Tigres template API allows one to programmatically create *workflows* using *templates* as the building blocks. These *templates* are composed of individual *tasks* that are units of work from the end-user that needs to be executed. A Tigres *workfow* is a python program (e.g. `my_program.py`) that uses the Tigres template API to build and execute a workflow *(Tigres program)*. There are four basic Tigres template functions: `sequence`, `parallel`, `merge` and `split`. The Tigres program can contain one or more of these template functions.

Execution behavior of the four Tigres templates



2.2 Core API

As mentioned previously, a Tigres program is a python program that uses the Tigres template API to build workflow. The templates are the core of the Tigres API. Any or all of the four Tigres templates can be used in a single program. A `Task` is the most basic unit of execution and can be defined as a python function internal to the Tigres program or a separate executable (e.g. `wget`, `my_c_program`). The figure above demonstrates the flow of execution with arrows. The inputs to each `Task` execution can be statically defined or retrieved from the results of previously executed *Tasks* or *Templates*.

A Glossary of Tigres Concepts

Task The atomic unit of execution. (see `tigres.Task`)

Task Array A collection of tasks. (see `tigres.TaskArray`)

Templates Patterns of execution from a combination of tasks. (see *Library Reference*)

Input Types The characteristic of the task the defines the type of the inputs. (see `tigres.InputTypes`)

Input Values The values used in a task execution. (see `tigres.InputValues`)

Input Array A collection of input values for a number of tasks. (see `tigres.InputArray`)

Each *template* function minimally takes two named collections: *Task Array* and *Input Array*. The *Task Array* is an ordered collection of tasks to be executed together, in sequence or parallel depending on the execution flow of the

particular template. The *InputArray* defines the inputs for each *Task* in the corresponding *Task Array* and is a collection of *Input Values*.

The *Task*, the atomic unit of execution in Tigres, has a collection of *Input Types* that specifies the type of inputs a task may take. A task's *Input Values* is an order list of task inputs and are passed to the task during execution. They are are not included in the task definition which allows for task reuse and late binding of data elements to the Tigres program execution. The data model in Tigres is described in greater detail in Section :ref:tigres-data-model-label

2.3 Monitoring API

The monitoring API has functions to:

- create monitoring information
- find and view monitoring information.

The monitoring information contains both information about template execution that is automatically generated by Tigres, and arbitrary user-provided information. All the monitoring information, both automatic and user-provided, is *semi-structured*, meaning it is broken into name/value pairs but only a few of the names and values are pre-defined. In general, the monitoring follows the Logging Best Practices that arose from the NetLogger project.

2.4 Execution Environments

Tigres can be executed in several different environments from batch queues to local threads and processes. By using the appropriate *execution engine*, a Tigres program can be executed on a single node or deployed without additional infrastructure to department clusters and batch processing queues on supercomputers. A program is written once and only the execution engine is changed at run time. This allows users to easily scale from development (desktop) to production (department clusters and HPC centers).

Tigres currently supports five execution engines.

- Local Threads - Tigres runs tasks as threads on one machine.
- Local Processes - Tigres runs tasks as processes on one machine.
- Distributed Processes - Tigres distributes tasks as processes across a cluster of machines.
- Sun Grid Engine - Tigres submits tasks as Sun Grid Engine jobs. This mode is used on HPC resources (e.g., NERSC) where a private instance of MySGE is run as a glidin.
- SLURM - Tigres submits tasks to a SLURM job manager

CHAPTER

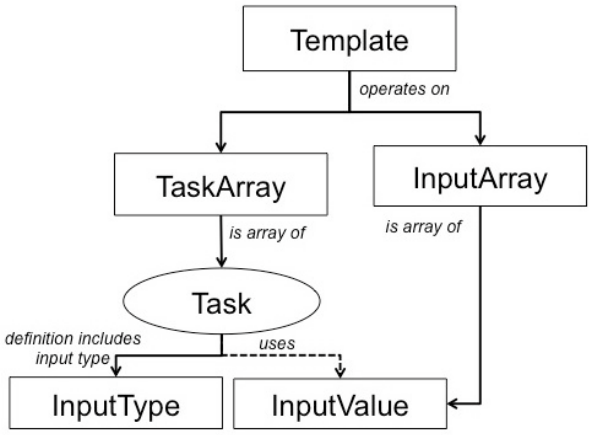
THREE

ARCHITECTURE

This section details the Tigres architecture. In this section, Tigres data types and their relationships with one another, a high-level overview of the major Tigres components and how they interact and the special Tigres syntax that allows you to explicitly specify data dependencies between workflow tasks is discussed.

3.1 Data Model

This section describes the Tigres data model concepts.



A *Template* takes a *TaskArray*, that is a collection of tasks to be executed together, and an *InputArray* with the corresponding inputs to the tasks. The elements in a TaskArray form a collection of tasks that need to execute either in a sequence or in parallel.

Each Task definition includes the InputTypes that defines the type of inputs that the task takes. The InputArray is a collection of InputValues. InputValues are the values that are inputs to the task and its types match what is defined in the InputTypes. InputValues are passed to the task during execution and not included in the task definition. This allows for reuse of task definitions and late binding of actual data elements to the workflow execution.

Tigres uses a special syntax called PREVIOUS syntax to create data dependencies between tasks. Using `tigres.PREVIOUS` the user can specify the output of a previously executed task as input for a subsequent task.

Not only can PREVIOUS create data dependencies between tasks, it can also span across templates.

Note: PREVIOUS can be both implicit and explicit. The semantics of PREVIOUS are affected by the template type.

3.1.1 Implicit PREVIOUS

PREVIOUS is implicit when there are no input values for a task execution. Below are the four Tigres templates with an explanation of how PREVIOUS is handled implicitly for each.

sequence

If input values for any of the tasks are missing, use the entire output of the previous task or template.

parallel

If the following conditions are met:

a) no inputs are given to the parallel task (*input_array*)

b) the results of the previous template or task is iterable

c) there is only one parallel task in the task_array

then each item of the iterable results becomes a parallel task.

split

If the split task is missing input values use the output from the previous task or template.

If parallel tasks are missing input, semantics of the *parallel* template are used.

merge

If merge task missing input values use all outputs of the previous parallel tasks.

If parallel tasks are missing input, semantics of the *parallel* template are used.

3.1.2 Explicit PREVIOUS

PREVIOUS is explicit when the following syntax is used as input in InputValues

- PREVIOUS
- PREVIOUS.i
- PREVIOUS.i[n]
- PREVIOUS.taskOrTemplateName
- PREVIOUS.taskOrTemplateName.i
- PREVIOUS.taskOrTemplateName.i[n]

Below are the valid usages of PREVIOUS. It must be passed as a single value in an InputValues and it is evaluated immediately before the execution of the Task it is paired with.

PREVIOUS

Use the entire output of the previous task as the input.

Valid Usage `InputValues([PREVIOUS])`

Validation

Did this task already run?
Are there results?

PREVIOUS.i

Used to split outputs across parallel tasks from the previous task.It matches the i-th output of the previous task/template to the i-th InputValues of the task to be run. This only works for parallel tasks.

Valid Usage `InputValues([PREVIOUS.i, PREVIOUS.i, ...])`

Validation

Did this task already run?
Are there results?

PREVIOUS.i[n]

Use the n-th output of the previous task or template as input.

Valid Usage `InputValues([PREVIOUS.i[n])`

Validation

Did this task already run?
Are there results?

PREVIOUS.task

Use the entire output of the previous template/task with specified name.

Valid Usage `InputValues([PREVIOUS.taskOrTemplateName])`

Validation

Does the named task or template exist?
Did this task already run?
Are there results?

PREVIOUS.task.i

Used to split outputs from the specified task or template across parallel tasks. Match the i-th output of the previous task/template to the i-th InputValues of the task to be run. This only works for parallel tasks.

Valid Usage `InputValues([PREVIOUS.taskOrTemplateName.i, PREVIOUS.taskOrTemplateName.i, ...])`

Validation

Does the named task or template exist?
Did this task already run?
Are there results?
Are the results indexable?

PREVIOUS.task.i[n]

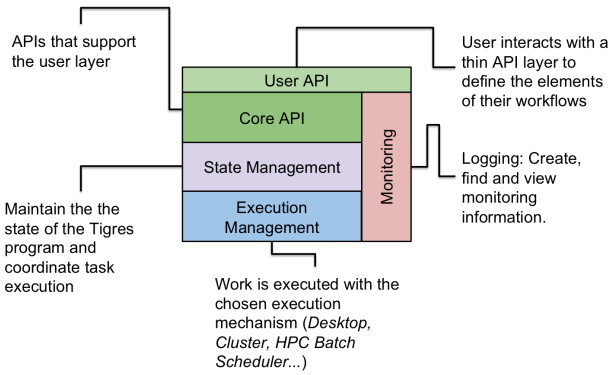
Use the n-th output of the previous task or template as input.

Valid Usage `InputValues([PREVIOUS.taskOrTemplateName.i[n]])`

Validation

Does the named task or template exist?
Did this task already run?
Are there results? Are the results indexable?

3.2 System Components



Tigres has five major components in a layered architecture: *User API*, *Core API*, *State Management*, *Execution Management* and *Monitoring*. The top most layer, the *User API*, is directly supported by *Monitoring* and *Core API* components. The execution type semantics is managed deeper down the stack by *State* and *Execution Management* components.

The user interacts with a thin User API to define the elements of their workflows, log messages and monitor their program execution. The core API is a collections of interfaces that support the User API with graphing, creating tasks, managing task dependencies and running templates. Monitoring is used by most other components to log system or user-defined events and to monitor the program with a set of entities for querying those events.

State management encompasses different execution management aspects of the workflow. For instance, it validates the user input to confirm they result in a valid workflow both prior to start as well as during execution. It transforms the Tigres tasks into *Work* that are then handed off to execution management. The state management layer also provides monitoring and provenance capabilities. It maintains state as each template is invoked and integrity of the running Tigres program.

The execution layer supports elastic resources and can work with one or more resource management techniques including HPC and cloud environments. In addition, the separation of the API and the execution layer allows us to leverage different existing workflow execution tools while providing a native programming interface to the user.

3.3 Monitoring

Monitoring information in Tigres is produced at two levels: system and user. All timestamped log events are captured in a single location (a file). The user is provided with an API for creating user-level events, checking for the status of tasks or templates and searching the logs with a special query syntax. System-level events provide information about the state of a program such as when a particular task started or what is it's latest status (e.g. did a task fail?).

3.3.1 User Level Logging

One of the design goals of the state management is to cleanly combine user-defined logs, e.g. about what is happening in the application, and the automatic logs from the execution system. The user-defined logging uses the familiar set of functions. Information is logged as a Python dict of key/value pairs. It is certainly possible to log English sentences, but breaking up the information into a more structured form will simplify querying for it later.

The following functions create log events at the specified log level. They are all wrappers around `tigres.write()`.

- `tigres.log_error()`
- `tigres.log_warn()`
- `tigres.log_info()`
- `tigres.log_debug()`
- `tigres.log_trace()`

Here is an example of creating trace level log events from within a function which details the input and output values.

```
def salutation(greeting, name):
    """

    :param greeting: A greeting
    :param name: The name to greet
    :return: personalized greeting or salutation
    """
    log_trace('input', function='salutation', message='greeting={}, name={}'.format(greeting, name))
    output = "{} {}".format(greeting, name)
    log_trace('output', function='salutation', message=output)
    return output
```

3.3.2 System Level Logging

The Tigres system may be monitored using `tigres.check()` and `tigres.query()`. A user may **check** the latest status of a program, task or template:

```
task_check = check('task', state=State.FAIL)
for task_record in task_check:
    print(".. State of {} = {} -  {}".format(task_record.name,
                                             task_record.state,
                                             task_record.errmsg))
```

Query gives the ability to filter log events by arbitrary key/value pair(s).:

```
# Check the logs for the decode event
log_records = query(spec=["event = input"])
for record in log_records:
    print(".. decoded {}".format(record.message))
```

3.4 Execution Management

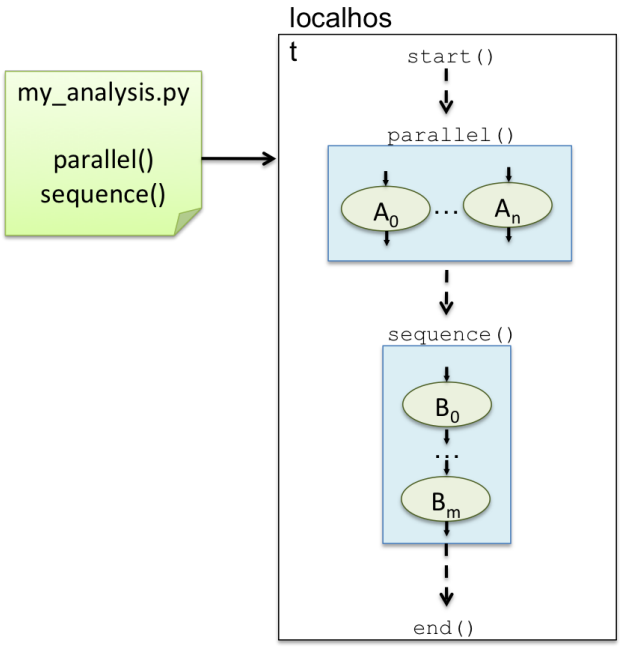
A Tigres program may be written once and run multiple times on different resource systems. The execution engine is specified at run time. A Tigres program can be executed on your personal desktop with local threads or processes. It can also be distributed across a cluster of nodes. Additionally, if you have access to a batch job scheduler, Tigres can use that. The interchangeability of different execution mechanisms allows you to develop your analysis on a desktop

or laptop and scale it up to a department clusters or HPC centers when your code is ready for production without actually changing the workflow code.

The different Tigres execution plugins require minimal setup. The plugins can be classified as desktop (single node), cluster and job manager execution plugins. Each execution plugin has its own mechanism for executing tasks as well as parallelization.

Type	Name	Execution Mechanism	Parallelization
Desktop			
	Local Thread	Threads	Local Worker Queue
	Local Process	Processes	Local Worker Queue
Cluster			
	Distribute Process	Processes	Task Server and Task Clients
Job Manager			
	SGE	SGE Job	Sun Grid Engine (`qsub`)
	SLURM	SLURM Job	SLURM (`squeue`)

3.4.1 Desktop Execution



There are two execution plugins for use on a single node: `LOCAL_THREAD` and `LOCAL_PROCESS`. Both plugins behave similarly except that `LOCAL_THREAD` spawns threads for each task while `LOCAL_PROCESS` spawns a process. The default Tigres execution is `LOCAL_THREAD`. You change the execution mechanism by passing in a keyword argument to `tigres.start()`:

```
from tigres import start
from tigres.utils import Execution
```

```
start(execution=Execution.LOCAL_PROCESS)
...
```

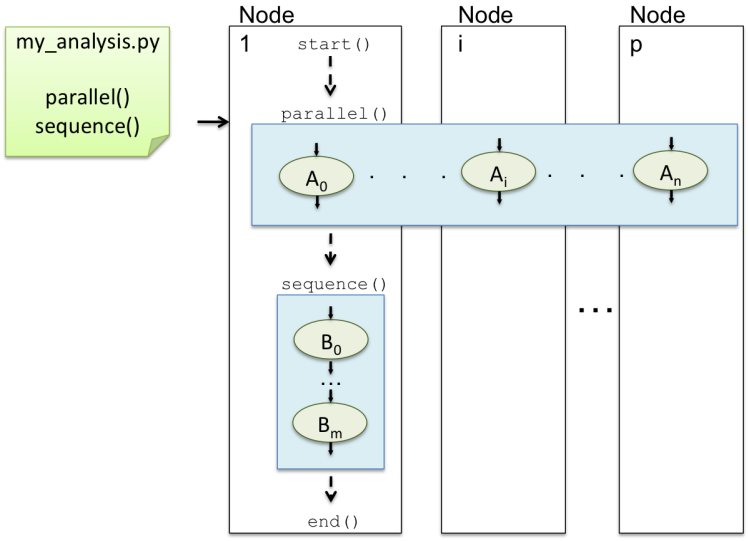
The figure to the right shows a simple Tigres program that launches *n* number of parallel tasks (A) and follows it up with a sequence of *m* tasks (B). This workflow is run on a single node (localhost).

Parallel execution is a very simple mechanism. All parallel tasks are placed in a worker queue, the number of processing cores is determined and then a worker is spawned for each processing core. In the case of *local thread* execution, the worker is a `threading.Thread` for *local process* execution a `multiprocessing.Process`.

3.4.2 Cluster Execution

Running the Tigres program from above on a set of nodes in a local cluster takes a little more set up. You change the execution plugin (`Execution.LOCAL_PROCESS`) and then set a one or more environment variables.

`DISTRIBUTE_PROCESS` distributes tasks across user specified host machines. A *TaskServer* is run in the main Tigres program. *TigresClient's*, which run workers that consume the tasks from the *TaskServer*, are launched on the hosts specified in `TIGRES_HOSTS` environment variable . If no host machines are specified, the *TaskServer* and a single *TaskClient* will be run on the local machine.



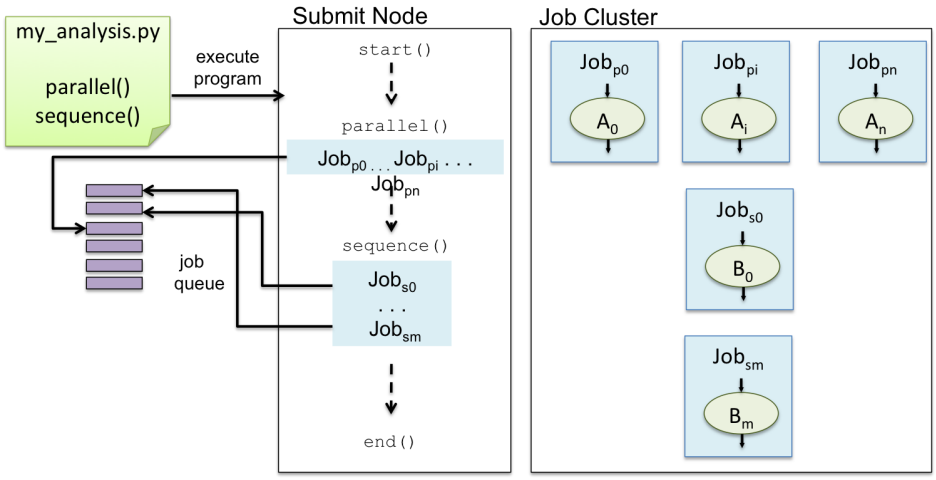
The Tigres program may have additional nodes configured with special environment variables: `TIGRES_HOSTS` and `OTIGRES_*`. `TIGRES_HOSTS` is a comma separated list of hosts names for distributing tasks. (i. e. `export TIGRES_HOSTS=n1,n2,n3`). Additionally, any environment variable prefixed with `OTIGRES_` will be passed to the *TaskClient* program on each node. For example, `OTIGRES_PATH=tigres_home/bin` will be translated to the command `PATH=tigres_home/bin; tigres-client` which is launched on the *TaskClient* nodes specified in `TIGRES_HOSTS`.

The figure above shows the same Tigres program as the previous figure. The program still has *n* parallel tasks and *m* sequential tasks but this time it is distributed across *p* nodes.:

```
localhost$ export TIGRES_HOSTS=node1,node2...nodep
localhost$ OTIGRES_PATH=tigres_home/bin my_analysis.py ...
```

The parallel template tasks are distributed across **Node 1** through **Node p**. The main Tigres program starts a *TaskServer* on **Node 1** and then launches *p TaskClients* on the nodes specified in `TIGRES_HOSTS`. The client programs run a process for each processing core until the *TaskServer* runs out of tasks. The sequence template tasks are run on **Node 1** where the main Tigres program is executing.

3.4.3 Job Manager Execution



If you have access to resources managed by a job scheduler, Tigres supports Sun Grid Engine (SGE) and Simple Linux Utility for Resource Management (SLURM). Each task is executed as a job on the batch queue. Parallelism is managed by the job manager. Tigres submits all the jobs at the same time and then waits for completion.

The requirements for using a job manager execution mechanism are that you must have an account and the ability to load the Tigres python module. Of course the correct execution plugin must be chosen:

```
from tigres import start
from tigres.utils import Execution
start(execution=Execution.SGE)
...
```

The naive assumption is that you would be able to run your Tigres program from the submission node and have unfettered access to the queue. At National Energy Research Scientific Computing Center (NERSC), this is achieved with a resource reservation that provides a personal installation of SGE called MySGE.

In the figure above, `my_analysis.py` is run at scale with managed resources. The program is run on the node where jobs are submitted to the batch queue. As with the other two examples, *Desktop Execution* and *Cluster Execution*, the workflow first runs a parallel template with *n* tasks and follows it up with a sequence template of *m* tasks. The parallel template submits *n* jobs simultaneously to the job manager. Upon their completion, the sequence template is executed. This template submits one task job at a time and waits until it completes before the next one is submitted.

CHAPTER

FOUR

TUTORIALS

Contents:

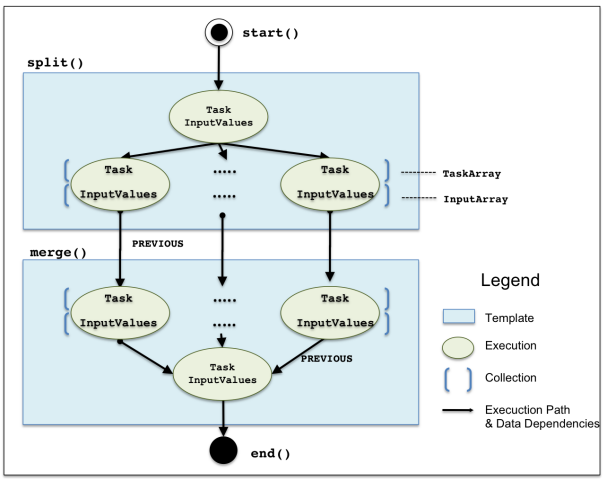
4.1 Basic Templates Tutorial

The Tigres Python API allows you to compose a workflow as a python application. Tasks may be an executable or python function. In this tutorial, you will learn how to a) set up your Tigres environment, b) use each Tigres template, c) define data dependencies between tasks, d) log messages and e) query the tigres logs. For the sake of simplicity, the tasks in this tutorial will be python functions.

This tutorial uses three programs which can be found here in a file named tigres-0.2.0-tutorials.zip . Each program has the same final output but uses different methods to produce it. The example workflows chosen for this tutorial perform string manipulation. The example functions are intentionally kept simple for allowing us to focus on the Tigres components in this tutorial. Real world examples can be plugged in similarly with any of the Tigres templates

Before starting the tutorials, follow the *Environment Setup* instructions to prepare your environment for Tigres workflow execution.

4.1.1 Data Model and Execution



The Figure to the left shows the relationship of the Tigres *Data Model* to the workflow execution. The *Split and Merge* template tutorial is used here to illustrate how the Tigres API functions and data structures relate to the execution flow of a Tigres program.

All Tigres programs may optionally begin with `start()` and finish with `end()` to initialize and finalize a Tigres program respectively. The core of this workflow is a `split()` template followed by a `merge()`. The (blue) boxes represent the templates and the (green) ovals inside are the `Task`s. The arrows simplistically signify the order of execution and data dependencies between `Task`s. The (blue) brackets indicate a collection (e.g. [Task .... Task] is collection of tasks called a TaskArray)

The first template executed is a `split` (top box). It requires four inputs: `split_task`, `split_input_values`, `task_array`, `input_array`. The first execution (top green oval) in this *template* is the `split_task` which takes `split_input_values` as input. Once this *task* completes it is followed by the parallel execution of the tasks in the *TaskArray*, called `task_array`, paired with each *InputValues* in the `input_array`.

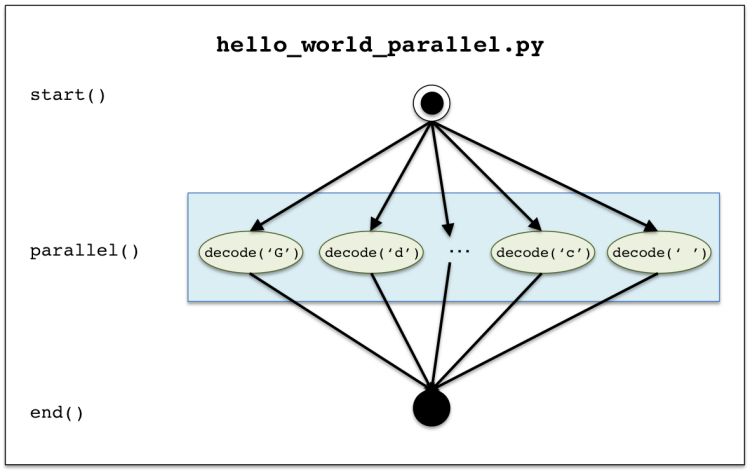
The second template to be executed is a `merge` (bottom blue box). A merge is the mirror image of the split. Its required inputs are: `task_array`, `input_array`, `merge_task`, `merge_input_values`. The parallel execution (`task_array`, `input_array`) happens first followed by a single merge task (`merge_task`, `merge_input_values`).

The data dependencies (blue dashed arrows) are specified by Tigres `PREVIOUS`. PREVIOUS is a syntax where the user can specify that the output of a previously executed task as input for a task. PREVIOUS creates dependencies between templates and can be either *implicit* or *explicit*.

4.1.2 Parallel

(a parallel template example)

First we will use `tigres.parallel()`. In this example (`hello_world_parallel.py`), which can be found in the zip file here, we will take a coded string containing a hidden message as input, decode the string using a parallel template and return the hidden message as output. Decoding the string with the parallel template takes advantage of the processing cores available to the Tigres program. If there is only one processing core, the string is decoded sequentially. However, if there are `P` processing cores where `P > 1`, then there will be `P` parallel tasks used to decode the string. This Tigres program as well as the others in this tutorial use a *Program Skeleton*.



The coded string, "Gdkkn\x1fShfqdr\x1fvnqkc ", in hello_world_parallel.py is of length *N* and is split into *N* tasks. There is one task for each character in the string. Each character will serve as input to a python function, decode(). This function will decode the string by adding one to the character's ASCII code. For example, "G" has an ASCII code of *71*. If we added one to make it *72*, the decoded character would be "H".

If you haven't done so already, follow the instructions in *Environment Setup* to setup your machine for running Tigres programs. If you haven't already downloaded the progams, download hello_world_parallel.py, to the machine you will be running your Tigres programs on. You are now ready to run your first Tigres Program. We will run the tasks as threads by passing EXECUTION_LOCAL_THREAD as the execution plugin.:

```
$ source $VIRTUALENV_HOME/envtigres/bin/activate
(envtigres)$ python hello_world_parallel.py \
     EXECUTION_LOCAL_THREAD
```

The output will looking similar to the following:

```
++++++++++++++++++++++++++++
Initial Input:
GdkknShfqdrvnqkc
++++++++++++++++++++++++++++

++++++++++++++++++++++++++++
Final Output:
Hello Tigres world!
++++++++++++++++++++++++++++
```

The *Final Output* you see is the decoded message, "Hello Tigres world!". Two output files are produced HelloWorldParallel.dot and hello_world_parallel.log:

```
$ ls
HelloWorldParallel.dot              hello_world_parallel.log        hello_world_parallel.py
```

The dot file which is discussed in the next section is a graph representation of the executed workflow. The log file is the detailed log of the Tigres workflow. The examples in this tutorial will detail how to write to the Tigres log, check the status of your workflow and query the logs for special information.

The next several sections will walk though *hello_world_parallel.py*.

Load Tigres API

In lines 7-8 the Tigres API is loaded into the python program namespace. Line 7 imports all functions and classes from tigres while line 8 imports the tigres.utils.Execution, tigres.utils.TigresException

and `tigres.utils.State`.

```
from tigres import *
from tigres.utils import Execution, TigresException, State
```

The main Function

The main function has some boiler plate code that is run before and after any templates are executed.

```
def main(execution):

    # Setup up the Tigres Program using __file__ to build the log file name
    start(log_dest=os.path.splitext(__file__)[0] + '.log', execution=execution)

    # Set the logging level
    set_log_level(Level.INFO)

    # Heart of Tigres Program here

     # Create DOT (plain text graph) file
    dot_execution()

    # End the Tigres Program
    end()
```

Program initialization

Before any templates are executed, the Tigres program is initialized with a call to `start()`. This function configures and initializes the Tigres program. In the above code, the program specifies the log file name (`log_dest`) as well as the execution mechanism (e.g. `execution='EXECUTION_LOCAL_THREADS'`). The function `start()` will end a previously started Tigres program. The verbosity of the logs is set using `set_log_level()`. After this call, all messages at a numerically higher level (e.g. DEBUG if level is INFO) will be skipped (See `Level`).

Graph visualization

After the last template is executed, the execution can be visualized using `dot_execution()`. This function creates a DOT file (plain text graph) for a visual representation of the completed workflow. The function `dot_execution()` writes a `graph file` of the current Tigres execution workflow that can be `visualized` using the visualization tools of your choice such as GraphViz. Lastly, the Tigres program is finalized with a call to `end()`.

The __main__ Driver

The driver program at the bottom of the file, does some argument checking, executes `main()` and prints any raised errors. If a `tigres.utils.TigresException` is raised, the message is printed and then the monitoring information is checked for more detailed error messages using `tigres.check()`. Line 26, searches the monitoring information for the currently running program for all Tigres tasks that have a state of `State.FAIL`.

```
if __name__ == "__main__":
    # Simple Usage Here
    if len(sys.argv) <= 1:
        print("Usage: {} ({})>".format(sys.argv[0], "|".join(Execution.LOOKUP.keys())))
        exit()
    try:
```

```
        main(Execution.get(sys.argv[1]))
    except TigresException as e:
        print(e)
        task_check = check('task', state=State.FAIL)
        for task_record in task_check:
            print(".. State of {} = {} -  {}".format(task_record.name, task_record.state, task_record
```

The Hidden Message

Look at the line 21 below the comment `#Heart of the Tigres Program` in the `main` function. Here the hidden message is declared as a string.

```
    # Heart of Tigres Program here
    hidden_message = 'Gdkkn\x1fShfqdr\x1fvnqkc '
    print("++++++++++++++++++++++++++++")
    print("Initial Input:")
    print (hidden_message)
```

The Decode Function

A function named `decode()` is defined. This function takes a single character value and decodes it by adding one to its ASCII value. The second to last line adds some debugging code. If the logging level is set to `DEBUG`, then the specified keyword value pairs will be written to the logs.

```
def decode(value):
    if isinstance(value, int):
        ordinal = value
    else:
        ordinal = ord(str(value))
    ordinal += 1
    log_debug("decode", message="{} to {}".format(value, unichr(ordinal)))
    return unichr(ordinal)
```

The workflow

In this section we will describe the main components of the Tigres program, `hello_world_parallel.py`. Start at line 27 after the declaration of `hidden_message`.

Decode Task

Lines 27-29 creates a task. The instantiated `tigres.Task` uses the `decode` function as its execution. The first parameter names the Task `"Decode"`. If the name is `None` then a unique name will be chosen by the Tigres framework. The second parameter declares this a task of type `FUNCTION` which means that a python function will be the third parameter. The other option is `EXECUTABLE` in which case a string representing an executable program (e. g. "wget", "my_c_program") would be specified. The `decode` function is passed as third parameter. The last parameter, `input_types`, defines the function arguments. It is an ordered list of data types.

```
    # Create Decode Tigres Task
    task_decode = Task("Decode", FUNCTION, impl_name=decode, input_types=[str])
```

Lines 30-32 create a `tigres.TaskArray` named *"Decode Tasks"*.

```
# Create a TaskArray with one our one decode task
task_array = TaskArray('Decode Tasks', tasks=[task_decode])
```

Decode Input

Line 33-35 defines what the input to the workflow is. Line 34 instantiates an empty `tigres.InputArray` named *"Coded Values"*.

```
# Create a Tigres InputArray of the character values to decode
input_array = InputArray("Coded Values", [])
```

Lines 36-40 iterates over the hidden message string and instantiates an `tigres.InputValues()` for each character in the string. Each `InputValues` is appended to `input_array` created on line 40.

```
# Iterate through the message string and add input values for each character
for c in hidden_message:
    input_values = InputValues("A Coded Value {}".format(c), [c])
    input_array.append(input_values)
```

Parallel Template

Line 42 has the call to `tigres.parallel()` which runs the decode function in parallel. The parallel template is named *"Fast Decoding"* and is passed the previously created `task_array` and `input_array`. The `parallel` template returns a python `list`. Each item in the list is the output from a parallel *"Decode"* task. Remember only one task was created but it was executed for each `InputValues` in the `InputArray`.

```
# Run the Tigres Parallel template
output = parallel('Fast Decoding', task_array, input_array)


# Check the logs for the decode event
log_records = query(spec=["event = decode"])
for record in log_records:
    print(".. decoded {}".format(record.message))
```

The final section of the code above uses the monitoring function `query()` to look for the log records written in the decode function. It searches for the keyword/value pair `event = decode`. However, this will only be logged when `set_log_level(Level.DEBUG)` (see below).

DEBUG Output

```
.. decoded G to H
.. decoded d to e
.. decoded k to l
.. decoded k to l
.. decoded n to o
.. decoded  to
.. decoded S to T
.. decoded h to i
.. decoded f to g
.. decoded q to r
.. decoded d to e
.. decoded r to s
.. decoded  to
.. decoded v to w
.. decoded n to o
```

```
.. decoded q to r
.. decoded k to l
.. decoded c to d
.. decoded   to !
```

Note: The `task_array` has one value while the `input_array` has many. The `parallel` template will interpret this as: *Run the task named "Decode" for each input value in the input_array*.

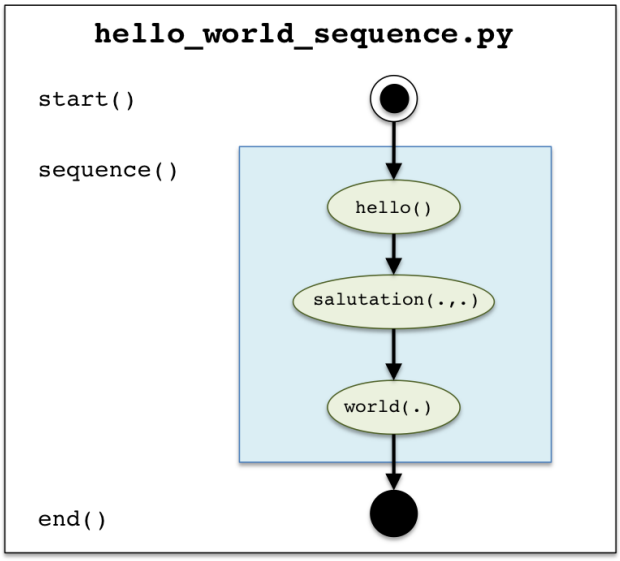
Finally, in line 54, the output list of the `parallel` template is converted to a string and written to standard output.

```
# Print the output
print("\n++++++++++++++++++++++++++++")
print("Final Output:")
print(''.join(output))
print("++++++++++++++++++++++++++++")
```

Note: The `tigres.parallel()` template returns a python `list` and it must be transformed to a string in order to read the decoded message.

4.1.3 Sequence

(a sequence template example)



The next example will use a `tigres.sequence()` template which has data dependencies. The data dependencies will be defined using the most basic form of `tigres.PREVIOUS` syntax. The full source example (`hello_world_sequence.py`) can be found in the zip file here.

This example builds the string `"Hello Tigres world!"` using `sequence()` with three tasks. The figure to the left represents the Tigres program `hello_world_sequence.py`. The arrows represent execution path and data dependencies. The first task runs `hello()` with no inputs. The second task in the sequence executes `salutation()` and it uses the output of `hello()` as its first input value. The third and last task invokes `world()` and takes the output of `salutation()` as input.

Download `hello_world_sequence.py`, to the machine you will be running your Tigres programs on. You will execute the tasks as processes by passing `EXECUTION_LOCAL_PROCESS` as the execution plugin.:

```
$ source $VIRTUALENV_HOME/envtigres/bin/activate
(envtigres)$ python hello_world_sequence.py \
    EXECUTION_LOCAL_PROCESS
```

The output you see will look something like the following:

```
++++++++++++++++++++++++++++
Initial Input:
None
++++++++++++++++++++++++++++


++++++++++++++++++++++++++++
Final Output:
Hello Tigres World!
++++++++++++++++++++++++++++
```

Two output files are produced `HelloWorldSequence.dot` and `hello_world_sequence.log`:

```
$ ls
HelloWorldSequence.dot              hello_world_sequence.log        hello_world_sequence.py
```

The `dot` file is a graph representation and the `log` file is the detailed log of the Tigres workflow.

The next several sections will walk though *hello_world_sequence.py*.

The Task Implementations

The `sequence` workflow has three tasks with three separate implementations. As mentioned previously, the implementations are python functions.

hello

The first function is very simple it takes no inputs and returns the string `"Hello"`. This is the first task in the workflow.

```
def hello():
    """

    :return: Hello
    :rtype: str
    """
    log_trace('input', function='hello', message='No inputs')
    output = "Hello"
    log_trace('output', function='hello', message=output)
    return output
```

salutation

The second function takes two string inputs and returns a personalized greeting. This is the second task in the workflow

```
def salutation(greeting, name):
    """

    :param greeting: A greeting
    :param name: The name to greet
```

```
    :return: personalized greeting or salutation
    """
    log_trace('input', function='salutation', message='greeting={}, name={}'.format(greeting, name))
    output = "{} {}".format(greeting, name)
    log_trace('output', function='salutation', message=output)
    return output
```

world

The third and final function task takes one string as input and returns the "Hello world" string. This is the last task in the workflow.

```
def world(salutation):
    """

    :param salutation: a greeting
    :return: Greeting to the world
    """
    log_trace('input', function='world', message='salutation={}'.format(salutation))
    output = "{} World!".format(salutation)
    log_trace('output', function='world', message=output)
    return output
```

The Workflow

Initialize the TaskArray and InputArray

Lines 63 through 67 add an empty `TaskArray` as well as an empty `InputArray`. They are populated later with the tasks and input values.

```
    # Create a TaskArray
    task_array = TaskArray('Hello World Tasks')

    # Create an InputArray
    input_array = InputArray("Hello World Inputs")
```

Hello Task

In lines 68-70, the first task is named `"Hello"` and assigned `hello()` as its implementation. This is appended to the `task_array` thus making it the first task in the sequence. Since the task function does not take any inputs no `input_types` are assigned to the task. However, an empty list must be appended to the `input_array`.

```
    # The Hello Task and its inputs
    task_array.append(Task("Hello", FUNCTION, impl_name=hello))
```

Warning: Order is important for `task_array` and `input_array`. The i th task in the task array is matched with the i th input in the input array.

Salutation Task

In line 74, the second task after "Hello" is named "Salutation" and assigned salutation() as its implementation. This is appended to the task_array making it the second task in the sequence workflow. Since this task takes two string inputs, input_types will be set to [str,str] which signifies that the first and second arguments to salutation are both strings.

```
# The Salutation Task and its inputs
task_array.append(Task("Salutation", FUNCTION, impl_name=salutation, input_types=[str, str]))
```

Line 75 defines the inputs to salutation(greeting, name). The first input is from the "Hello" task and the second will be the string literal, "Tigres". The output from the "Hello" task is specified as the first argument with PREVIOUS. This says, "Get the output of the task that was executed previous to this one and use it as the first input".

```
input_array.append([PREVIOUS, "Tigres"])
```

World Task

The final task in lines 76 through 78, named "World", is assigned world() as its implementation. It takes one string as input from the previous "Salutation" task.

```
# The World Task and its inputs
task_array.append(Task("world", FUNCTION, impl_name=world, input_types=[str]))
input_array.append([PREVIOUS])
```

Sequence Template

Lines 80-91 has the call to tigres.sequence() which runs the sequence template. It is named *"Hello World"* and passed in the previously created task_array and input_array.

```
# Run the Tigres Sequence template
output = sequence('Hello World', task_array, input_array)

# Check the logs for the decode event
log_records = query(spec=["{} ~ .*put".format(Keyword.EVENT)])
for record in log_records:
    print(".. function: {}, {}: {}".format(record["function"], record.event, record.message))
```

The final section of the code above uses the monitoring function query() to look for the log records written in the task functions. It searches for records matching the following condition: event ~ .*put. The ~ symbol compares the string value, event , with the regular expression .*put and returns all matches. However, this will only be logged when set_log_level(Level.TRACE) (see below).

TRACE Output

```
.. function: hello, input: No inputs
.. function: hello, output: Hello
.. function: salutation, input: greeting=Hello, name=Tigres
.. function: salutation, output: Hello Tigres
.. function: world, input: salutation=Hello Tigres
.. function: world, output: Hello Tigres World!
```

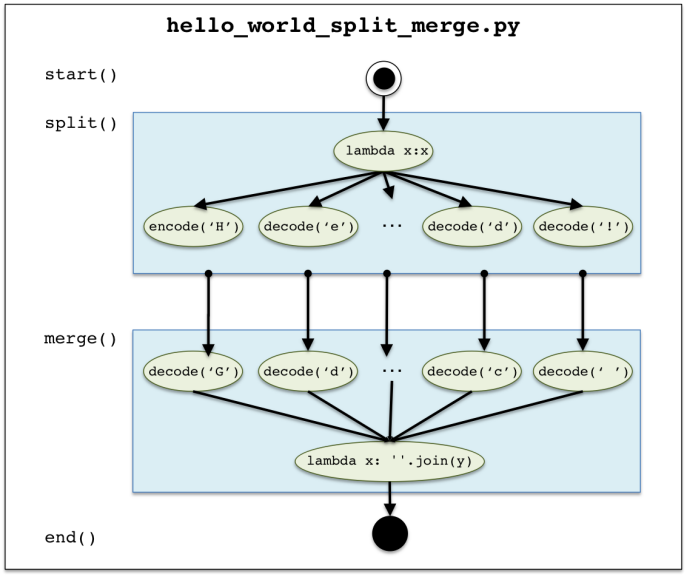
Finally, the output is written to standard output (Line 96).

```
    # Print the output
    print("\n++++++++++++++++++++++++++++")
    print("Final Output:")
    print(output)
    print("++++++++++++++++++++++++++++")
```

4.1.4 Split and Merge

(a example using split and merge templates)

This last example covers both the `tigres.split()` and `tigres.merge()` templates and how to take advantage of the implicit data dependency mechanism between tasks in Tigres. The full source of the program (`hello_world_split_merge.py`) can be found in the zip file here.



This example is based on the *Parallel* example. Instead of decoding a hidden message, it takes a clear-text message, `"Hello World Tigres!"`, encodes it using a `tigres.split()` template and then decodes it using a `tigres.merge()` template. The figure to the right represents the Tigres program `hello_world_split_merge.py`. Notice that there are two (blue) rectangles representing the use of two Tigres templates. The first one is `split()` and the latter `merge()`. The arrows show the execution path and data dependencies. Also, there are additional task implementations. In addition to `decode()` from the *Parallel* example, `encode()` and two `lambda` functions have been added.

Download `hello_world_split_merge.py`, to the machine you will be running your Tigres programs on. You will execute the tasks as processes by passing `EXECUTION_LOCAL_PROCESS` as the execution plugin.:

```
$ source $VIRTUALENV_HOME/envtigres/bin/activate
(envtigres)$ python hello_world_split_merge.py \
    EXECUTION_LOCAL_PROCESS
```

The output you see will look something like the following:

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Initial Input:
Hello Tigres world!
++++++++++++++++++++++++++
```

```
++++++++++++++++++++++++++++
Intermediate Output:
[u'G', u'd', u'k', u'k', u'n', u'\x1f', u'S', u'h', u'f', u'q', u'd', u'r', u'\x1f', u'v', u'n', u'q
++++++++++++++++++++++++++++


++++++++++++++++++++++++++++
Final Output:
Hello Tigres world!
++++++++++++++++++++++++++++
```

Two output files are produced `HelloWorldSplitMerge.dot` and `hello_world_split_merge.log`:

```
$ ls
HelloWorldSplitMerge.dot            hello_world_split_merge.log     hello_world_split_merge.py
```

The `dot` file is a graph representation and the `log` file is the detailed log of the Tigres workflow.

The next several sections will walk through *hello_world_split_merge.py*.

The Task Functions

This `encode()` function takes a character, subtracts one from the ASCII value and returns the new ASCII character.

```
def encode(value):
    if isinstance(value, int):
        ordinal = value
    else:
        ordinal = ord(str(value))
    ordinal -= 1
    log_debug("encode", message="{} to {}".format(value, unichr(ordinal)))
    return unichr(ordinal)
```

This `decode()` function takes a character, adds one from the ASCII value and returns the new ASCII character.

```
def decode(value):
    if isinstance(value, int):
        ordinal = value
    else:
        ordinal = ord(str(value))
    ordinal += 1
    log_debug("decode", message="{} to {}".format(value, unichr(ordinal)))
    return unichr(ordinal)
```

Encoding using a Split Template

The string `"Hello Tigres World!"` is encrypted in lines 26 through 54 of `hello_world_split_merge.py` (see below). This function uses a split template. In a split template, a single task (split task) and it's corresponding inputs as well as a task array and its corresponding input array are required. The split task is executed first and prepares the inputs for the task array that executes tasks in parallel after the split task is finished.

```
1       # ########################################################
2       # Setting up a Split Template for encoding the message
3       ########################################################
4
5       # Create a Task for parsing the string
```

```
6       task_to_list = Task("String to List", FUNCTION,
7                           impl_name=lambda x: x, input_types=[str])
8       input_string = InputValues("String to encode", [message])
9
10      # Create a Task Array for the encode_string tasks
11      task_encode = Task("Encode", FUNCTION, impl_name=encode, input_types=[str])
12      task_array_encode = TaskArray('Encoding Tasks', tasks=[task_encode])
13
14      # We create an empty input array that will be implicitly filled with the
15      # PREVIOUS  task's output. The output of the of the split task must be iterable
16      input_array_encode = InputArray("String Values", [])
17      tmpoutput = split("Fast Encoding", task_to_list,
18                        input_string, task_array_encode, input_array_encode)
19
20      # Print the output
21      print("\n++++++++++++++++++++++++++++")
22      print("Intermediate Output:")
23      print(tmpoutput)
24      print("++++++++++++++++++++++++++++\n")
```

- Lines 6-8 above create the split task and its input. The task is defined as a python function that takes a string. Notice here that a `lambda` function is used for the implementation. It simply returns the given input. No special processing of the string is necessary. The idea it to use the split template's implicit ability to split a generator object (i.e. `list`) into N parallel tasks.
- Lines 11-12 create the task array and its task definition. Here only one task `"Encode"` is specified with `encode()` as its implementation. This is because the split template semantics will determine how many copies of this task will be run at execution time from the input string.
- Line 16 specifies and empty input array. This tells the split template to go ahead and determine the input from the split task output.
- Finally, Lines 17 and 18 execute the split template named `"Fast Encoding"`

Decoding using a Merge Template

In lines 56 through 85 of `hello_world_split_merge.py`, the results of the `split()` template will be decoded by the `merge()` template. In a merge template, a single task (merge task) and it's corresponding inputs as well as a task array and its corresponding input array are required.

```
1       ########################################################
2       # Setting up a Merge Template for decoding the message
3       ########################################################
4
5       # Create Decode Tigres Task
6       task_decode = Task("Decode", FUNCTION, impl_name=decode, input_types=[str])
7       # Create a TaskArray with one our one decode_string task
8       task_array = TaskArray('Decoding Tasks', tasks=[task_decode])
9       # Run the Tigres Split template
10      output = merge('Fast Decoding', task_array, None,
11                     Task("To String", FUNCTION, impl_name=lambda x: ''.join(x)))
12
13      # Print the output
14      print("\n++++++++++++++++++++++++++++")
15      print("Final Output:")
16      print output
17      print("++++++++++++++++++++++++++++\n")
18
```

```
19      log_records = query(spec=["event ~ ..code"])
20      for record in log_records:
21          print(".. {}d {}".format(record.event, record.message))
22
23      # Create DOT (plain text graph) file
24      dot_execution()
```

Three statements are executed:

- Line 6 creates a "Decode" task with decode() as its implementation
- Line 8 creates a task array with one task definition because merge will determine the number of decode tasks to execute.
- Lines 10 and 11 is the execution of the merge template. None is passed in for the input array (line 47) to invoke implicit data dependency between the split and merge templates. The merge template will try to get the results from the previously executed template. The merge task python implementation is a lambda() function that joins the characters into a single string. No input values are specified. The merge template will attempt to retrieve the results from the execution of the task array.
- Line 16, the output list of the parallel template written to standard output.
- Line 19-21 uses the monitoring function query() to look for the log records written in the task functions. It searches for records matching the following condition: event ~ .*put. The ~ symbol compares the string value, event , with the regular expression .*put and returns all matches. However, this will only be logged when set_log_level(Level.DEBUG) (see below).

DEBUG Output:

```
.. encoded H to G
.. encoded e to d
.. encoded l to k
.. encoded l to k
.. encoded o to n
.. encoded   to
.. encoded i to h
.. encoded T to S
.. encoded g to f
.. encoded r to q
.. encoded e to d
.. encoded s to r
.. encoded   to
.. encoded w to v
.. encoded o to n
.. encoded r to q
.. encoded l to k
.. encoded d to c
.. encoded ! to
.. decoded G to H
.. decoded d to e
.. decoded k to l
.. decoded k to l
.. decoded n to o
.. decoded  to
.. decoded S to T
.. decoded h to i
.. decoded f to g
.. decoded q to r
.. decoded d to e
.. decoded r to s
```

```
.. decoded  to
.. decoded v to w
.. decoded n to o
.. decoded q to r
.. decoded k to l
.. decoded c to d
.. decoded   to !
```

4.2 HPC Tutorial

Scaling up a Tigres workflow to an HPC system like those at NERSC is very simple. This tutorial will demonstrate how to run a Tigres workflow at NERSC and assumes that you have a nersc account. It will walk you through setting up a python environment for execution and submitting Simple Linux Utility for Resource Management (SLURM) script to both of the NERSC systems: Edison and Cori.

4.2.1 Python Environment

The following steps detail how to setup a python environment for running Tigres worklflows on NERSC resources. The instructions are for edison but should work with other similary configured NERSC resources.

The following set of instructions assume:

- You have a NERSC account
- You want to submit jobs to your default NERSC repo

1. **Get Latest Tigres Release** Download Tigres 0.2.0 source distribution here. Now copy this to your NERSC home directory:

   ```
   $ scp tigres-0.2.0.tar.gz [user_name]@dtn01.nersc.gov:./
   ```

2. **Get Tutorials Archive** Download Tigres 0.2.0 tutorials here. Now copy this to your NERSC home directory:

   ```
   $ scp tigres-0.2.0-tutorials.zip [user_name]@dtn01.nersc.gov:./
   ```

   Login to NERSC and unzip the tutorials in your home directory:

   ```
   $ ssh [user_name]@edison.nersc.gov
   $ unzip  tigres-0.2.0-tutorials.zip
   $ cd tigres-0.2.0-tutorials
   ```

3. **Setup Script** Change to the tutorial directory directory and run `setup_env.sh`. The setup script prepares the Tigres python Environment. After running the script there will be a new virtualenv environment, `env<NERSC_HOST>`, that is suffixed with the NERSC host name.:

   ```
   $ ./set_env.sh
   ```

4. **Install Tigres** There is one final step. Tigres must now be installed into the virtualenv environment that was setup in the previous step.

```
$ module load python
$ source env$NERSC_HOST/bin/activate
(envedison)$ pip install --no-index $HOME/tigres-0.2.0.tar.gz
```

4.2.2 Basic Statistics Example

Once the tigres environment is setup in your NERSC home directory, you are ready to run the sample program, `basic_statistics_by_column.py`, which takes a delimited text file and performs basic statistics (total, mean, median, variance, standard deviation) on the specified columns. The Tigres program extracts the data and uses a parallel template for the statistical calculations.

This section will first walk you through the steps to get the sample data and submit the job to the job manager queue.

---

1. **Get the Data** This example uses a large dataset, household_power_consumption.txt, from the UC Irvine Machine Learning repository.:

   ```
   $ wget https://archive.ics.uci.edu/ml/machine-learning-databases/00235/household_power_consu
   $ unzip household_power_consumption.zip
   ```

2. **Submit the Job** A SLURM script called, `tigres_run.slurm`, has been provided in the tutorial archive. This script is used to submit the Tigres program to the NERSC queue.

   Submit the job to SLURM:

   ```
   $ sbatch tigres_run.slurm
   Submitted batch job 1031378
   ```

   The job is now submitted and should run in the debug queue:

   ```
   $ squeue -u \[username\]
        JOBID       USER  ACCOUNT       NAME  PARTITION    QOS NODES   TIME_LIMIT       TIME
      1031378        <username>   <repo>   hconsume      debug normal     2        30:00
   ```

3. Check the Results Once the SLURM job is finished, you may check the results in the output file:

   ```
   $ cat slurm_<job>.out
   /global/u2/v/vch/tigres-0.1.1-tutorials:/global/u2/v/vch/tigres-0.1.1-tutorials/envedison/bin:/g
   TIGRES_HOSTS nid00185,nid00186
   results - ('average_by_column', 'Global_intensity', 4.6277593105838)
   results - ('total_by_column', 'Global_intensity', 9483574.59999317)
   results - ('median_by_column', 'Global_intensity', 2.6)
   results - ('stdev_by_column', 'Global_intensity', 4.444396259786258)
   results - ('variance_by_column', 'Global_intensity', 19.752658114002077)
   results - ('average_by_column', 'Voltage', 240.83985797447758)
   results - ('total_by_column', 'Voltage', 493548304.1499374)
   results - ('median_by_column', 'Voltage', 241.01)
   results - ('stdev_by_column', 'Voltage', 3.23998667900864)
   results - ('variance_by_column', 'Voltage', 10.497513680153437)
   results - ('average_by_column', 'Global_reactive_power', 0.12371447630385488)
   results - ('total_by_column', 'Global_reactive_power', 253525.60199996372)
   results - ('median_by_column', 'Global_reactive_power', 0.1)
   results - ('stdev_by_column', 'Global_reactive_power', 0.11272197955071389)
   results - ('variance_by_column', 'Global_reactive_power', 0.012706244673831562)
   results - ('average_by_column', 'Global_active_power', 1.091615036500693)
   results - ('total_by_column', 'Global_active_power', 2237024.86200014)
   results - ('median_by_column', 'Global_active_power', 0.602)
   results - ('stdev_by_column', 'Global_active_power', 1.0572941610939552)
   ```

```
    results - ('variance_by_column', 'Global_active_power', 1.1178709430833706)
    results - ('average_by_column', 'Sub_metering_3', 6.45844735712055)
    results - ('total_by_column', 'Sub_metering_3', 13235167.0)
    results - ('median_by_column', 'Sub_metering_3', 1.0)
    results - ('stdev_by_column', 'Sub_metering_3', 8.437153908665618)
    results - ('variance_by_column', 'Sub_metering_3', 71.18556607851151)
    results - ('average_by_column', 'Sub_metering_2', 1.2985199679887571)
    results - ('total_by_column', 'Sub_metering_2', 2661031.0)
    results - ('median_by_column', 'Sub_metering_2', 0.0)
    results - ('stdev_by_column', 'Sub_metering_2', 5.822026473177329)
    results - ('variance_by_column', 'Sub_metering_2', 33.895992254377646)
    results - ('average_by_column', 'Sub_metering_1', 1.1219233096502186)
    results - ('total_by_column', 'Sub_metering_1', 2299135.0)
    results - ('median_by_column', 'Sub_metering_1', 0.0)
    results - ('stdev_by_column', 'Sub_metering_1', 6.153031089701269)
    results - ('variance_by_column', 'Sub_metering_1', 37.85979159083039)
    ./basic_statistics_by_column.py EXECUTION_DISTRIBUTE_PROCESS household_power_consumption.txt ';'
```

4.2.3 SLURM Job Script

The SLURM script, `tigres_run.slurm`, can be submitted on any NERSC system. This script uses the Tigres distribute plugin to execute the basic statistics script on two nodes.

- *Lines 3-7:* Sets up the SLURM job (i.e specifies name, number cores ..)
- *Lines 9-11:* Loads the Tigres environment
- *Lines 13-17:* Setups up environment variables
- *Lines 30-34:* Prepares the distrbute plugin special environment variables
- *Line 35:* Executes the Tigres workflow

```
1  #!/bin/sh
2
3  #SBATCH -p debug
4  #SBATCH --ccm
5  #SBATCH -N 2
6  #SBATCH -t 00:30:00
7  #SBATCH -J hconsume
8
9  # Load the tigres python environment
10 module load python
11 source env$NERSC_HOST/bin/activate
12
13 # Add the application code to the paths
14 export PYTHONPATH=$SLURM_SUBMIT_DIR:$PYTHONPATH
15 export PATH=$SLURM_SUBMIT_DIR:$PATH
16 echo $PATH
17 cd $SLURM_SUBMIT_DIR
18
19 # Determine the hosts available. Convert the compact host list to a
20 # comma separated list.
21 export TIGRES_HOSTS=`scontrol show hostname $SLURM_JOB_NODELIST  | awk -vORS=, '{ print $1 }' | sed s
22 echo "TIGRES_HOSTS ${TIGRES_HOSTS}"
23
24
25 # The workflow is executed with the Tigres Distribute plugin.
26 # Since the tasks will be executed across nodes, we need to
```

```
27 # define the environment with OTIGRES_* environment
28 # variables.  Also, TIGRES_HOSTS will list all
29 # the hosts used for this workflow
30 export OTIGRES_PATH=$PATH
31 export OTIGRES_PYTHONPATH=$SLURM_SUBMIT_DIR/env$NERSC_HOST/lib/python2.7/site-packages:$PYTHONPATH
32 export OTIGRES_LD_LIBRARY_PATH=$LD_LIBRARY_PATH
33 export EXECUTION=EXECUTION_DISTRIBUTE_PROCESS
34 ./basic_statistics_by_column.py EXECUTION_DISTRIBUTE_PROCESS household_power_consumption.txt ';' 2,3,
35 echo "./basic_statistics_by_column.py EXECUTION_DISTRIBUTE_PROCESS household_power_consumption.txt '
```

CHAPTER

FIVE

ENVIRONMENT SETUP

In order to write and run a Tigres program, the Tigres Python API needs to be installed. We recommend setting up a Tigres python environment with virtualenv. The following instructions assume that you have at least Python 2.7.x, setuptools and virtualenv installed on your system. If you don't have these installed, here are some pointers to get these installed:

5.1 Install Setuptools

If you don't have setuptools installed, choose the instructions below for your machine architecture:

- Unix
- Mac-Os

5.2 Install Virtualenv

Once you have setuptools installed, you can easily install any python module with the *easy_install* command. Now install virtualenv:

```
sudo easy_install virtualenv
```

5.3 Install Tigres API

Installing Tigres involves creating a virtualenv environment for Tigres development and then installing Tigres.

5.3.1 Setup Tigres VirtualEnv

In order to create a virtualenv environment for Tigres, you need to set up a directory that contains all virtualenv environments, if you don't already have one:

```
$ export VIRTUALENV_HOME=$HOME/.virtualenv
$ mkdir $VIRTUALENV_HOME
$ virtualenv $VIRTUALENV_HOME/envtigres
```

5.3.2 Download and install Tigres

The last step is to activate the Tigres virtualenv environment you just created and install the Tigres API. Download Tigres 0.2.0 source distribution here. For Python 2.7 you may need to install cloud first (pip install cloud).

```
$ source $VIRTUALENV_HOME/envtigres/bin/activate
(envtigres)$ cd <download_dir>
(envtigres)$ pip install --no-index tigres-0.2.0.tar.gz
```

CHAPTER

SIX

USEFUL RESOURCES

6.1 Quick Reference

6.1.1 Basic steps to create a workflow

(1) Define input types and values for tasks **(2)** Create tasks
(3) Create (and run) template with tasks and inputs and use output from template for next stage of workflow.

6.1.2 Data Types

`Task`: Function or program; atomic unit of execution
`TaskArray`: List of one or more Tasks, which will be executed in a Template
`InputTypes`: Types for inputs of a Task.
`InputValues`: Values for inputs of a Task.
`InputArray`: One or more InputValues, which will be inputs to a TaskArray in a Template

Templates:

`sequence`: List of tasks placed in series; output of one is the input of the next
`parallel`: List of tasks processing their inputs in parallel
`split`: "split" task feeding inputs to a set of parallel tasks
`merge`: Collect output of parallel tasks

6.1.3 Functions

Note: Name is an optional parameter in all below functions.

(1) Create inputs for tasks

Description	Functions and examples
Create InputTypes	tigres.InputTypes `input_type_1 = InputTypes("Types1", [int, str])`
Create InputValue	tigres.InputValues `input_value_1 = InputValues("Values1", [1, "hello"])`
Create InputArray	tigres.InputArray `input_array_12 = InputArray("Array12", [input_value_1, input_value_2])`

(2) Create tasks and task arrays

Description	Functions and examples

Create a Task	tigres.Task
	`task_f1 = Task("A", FUNCTION, "myfunc", input_type_1)`
	`task_x1 = Task("X", EXECUTABLE, "./bin/x", input_type_1)`

Create a TaskArray	tigres.TaskArray
	`task_array_xyz = TaskArray("xyz", [task_f1, task_x1])`
	`task_array_12 = TaskArray("tasks12", [task_f1, task_f2])`

(3) Create templates

Description	Functions and examples
Sequence	`tigres.sequence()` `output = sequence("my_sequence", task_array_12, input_array_12)`
Data parallel	`tigres.parallel()` `output = parallel("abc", task_array_12, input_array_12)`
Split	`tigres.split()` `output = split("split", task_x1, input_value_1, spl_taskarray, spl_input_array)`
Merge	`tigres.merge()` `output = merge("sync", syn_ta, syn_ia, task_x1, input_value_1)`

Constants and Special Notations

Constant	Description
FUNCTION	Function in current program
EXECUTABLE	Different program to execute
PREVIOUS.task_name.i[output_no]	Output number of previous task task_name

6.2 Program Skeleton

Below is a skeleton to help kickstart writing your Tigres code. This will allow you to specify different Execution plugins. You can download the Tigres skeleton program here

```
"""
My Tigres Program
"""
import os
import sys

from tigres import *
from tigres.utils import Execution, TigresException, State


def main(execution):

    # Setup up the Tigres Program using __file__ to build the log file name
```

```
14      start(log_dest=os.path.splitext(__file__)[0] + '.log', execution=execution)
15
16      # Set the logging level
17      set_log_level(Level.INFO)
18
19      # Heart of Tigres Program here
20
21       # Create DOT (plain text graph) file
22      dot_execution()
23
24      # End the Tigres Program
25      end()
26
27  if __name__ == "__main__":
28      # Simple Usage Here
29      if len(sys.argv) <= 1:
30          print("Usage: {} ({})>".format(sys.argv[0], "|".join(Execution.LOOKUP.keys())))
31          exit()
32      try:
33          main(Execution.get(sys.argv[1]))
34      except TigresException as e:
35          print(e)
36          task_check = check('task', state=State.FAIL)
37          for task_record in task_check:
38              print(".. State of {} = {} -  {}".format(task_record.name, task_record.state, task_record
```

This program allows you to execute your workflow with any of the available execution plugins. If you execute this python code, you will see all the available execution plugins:

```
$ python hello_world_parallel.py
Usage: hello_world_parallel.py (EXECUTION_SLURM|EXECUTION_DISTRIBUTE_PROCESS|EXECUTION_LOCAL_THREAD|E
```

Execute the script with `EXECUTION_LOCAL_THREAD`:

```
$ hello_world_parallel.py EXECUTION_LOCAL_THREAD
```

There is no output to the screen but you will see a log file in the current directory:

```
$ ls
hello_world_parallel.log    hello_world_parallel.py
```

6.2.1 Load Tigres API

In lines 7-8 the Tigres API is loaded into the python program namespace. Line 7 imports all functions and classes from `tigres` while line 8 imports the `tigres.utils.Execution`, `tigres.utils.TigresException` and `tigres.utils.State`.

```
from tigres import *
from tigres.utils import Execution, TigresException, State
```

6.2.2 The main Function

The main function has some boiler plate code that is run before and after any templates are executed.

```
def main(execution):

    # Setup up the Tigres Program using __file__ to build the log file name
    start(log_dest=os.path.splitext(__file__)[0] + '.log', execution=execution)

    # Set the logging level
    set_log_level(Level.INFO)

    # Heart of Tigres Program here

     # Create DOT (plain text graph) file
    dot_execution()

    # End the Tigres Program
    end()
```

Program initialization

Before any templates are executed, the Tigres program is initialized with a call to `start()`. This function configures and initializes the Tigres program. In the above code, the program specifies the log file name (`log_dest`) as well as the execution mechanism (e.g. `execution='EXECUTION_LOCAL_THREADS'`). The function `start()` will end a previously started Tigres program. The verbosity of the logs is set using `set_log_level()`. After this call, all messages at a numerically higher level (e.g. DEBUG if level is INFO) will be skipped (See `Level`).

Graph visualization

After the last template is executed, the execution can be visualized using `dot_execution()`. This function creates a DOT file (plain text graph) for a visual representation of the completed workflow. The function `dot_execution()` writes a `graph file` of the current Tigres execution workflow that can be `visualized` using the visualization tools of your choice such as GraphViz. Lastly, the Tigres program is finalized with a call to `end()`.

6.2.3 The __main__ Driver

The driver program at the bottom of the file, does some argument checking, executes `main()` and prints any raised errors. If a `tigres.utils.TigresException` is raised, the message is printed and then the monitoring information is checked for more detailed error messages using `tigres.check()`. Line 26, searches the monitoring information for the currently running program for all Tigres tasks that have a state of `State.FAIL`.

```
if __name__ == "__main__":
    # Simple Usage Here
    if len(sys.argv) <= 1:
        print("Usage: {} ({})>".format(sys.argv[0], "|".join(Execution.LOOKUP.keys())))
        exit()
    try:
        main(Execution.get(sys.argv[1]))
    except TigresException as e:
        print(e)
        task_check = check('task', state=State.FAIL)
        for task_record in task_check:
            print(".. State of {} = {} -  {}".format(task_record.name, task_record.state, task_record
```

CHAPTER

SEVEN

LIBRARY REFERENCE

This is the main package, containing the API to create and run templates; and perform and query monitoring and provenance. For background information on what Tigres is meant to do, see the *Concepts in Tigres*. The top-level `tigres` module has tools for initializing and finalizing tigres programs. This can be done explicitly using `tigres.start()` and `tigres.end()` or implicitly when the first Tigres data object is initialized.

7.1 *tigres*

platform Unix, Mac

synopsis The high level Tigres API. Initialization of Tigres templates, tasks and their inputs.

module author Dan Gunter <dkgunter@lbl.gov>, Gilberto Pastorello <gzpastorello@lbl.gov>, Val Hendrix <vhendrix@lbl.gov>

Example

```
>>> import tempfile as temp_file
>>> from tigres import *
>>> f = temp_file.NamedTemporaryFile()
>>> program=start(log_dest=f.name)
>>> set_log_level(Level.INFO)
>>> log_info("Tasks", message="Starting to prepare tasks and inputs")
>>> def adder(a, b): return a + b
>>> task1 = Task("Task 1", FUNCTION, adder)
>>> task2 = Task("Task 2", EXECUTABLE, "/bin/echo")
>>> tasks = TaskArray(None, [task1, task2])
>>> inputs = InputArray(None, [InputValues(None, [2, 3]), InputValues(None, ['world'])])
>>> log_info("Template", message="Before template run")
>>> sequence("My Sequence", tasks, inputs)
'world'
```

```
>>> # find() returns a list of records
>>> for o in find(activity="Tasks"):
...     assert o.message
>>> end()
```

7.1.1 Templates

- sequence()
- parallel()
- parallel_sequential()
- split()
- merge()

7.1.2 Tasks and Inputs

- InputArray - Array of one or more InputValues, which will be inputs to a TaskArray in a Template.
- InputTypes - List of types for inputs of a Task.
- InputValues - List of values for inputs matched to a task Task.
- Task - Function or program. A task is the atomic unit of execution in Tigres.
- TaskArray - List of one or more Tasks, which will be executed in a Template
- EXECUTABLE - identifies that a task implementation is an executable
- FUNCTION - identifies that a task implementation is a function

7.1.3 Initialization

Users should use start() to initialize the Tigres library, and this will initialize the logging as well. When start() is called, the log_dest keyword in that function is passed as the first argument logger and any other keywords beginning in log_ will have the prefix stripped and will be passed as keywords.

- start()
- end()
- get_results()
- set_log_level()

7.1.4 User logging

One of the design goals of the state management is to cleanly combine user-defined logs, e.g. about what is happening in the application, and the automatic logs from the execution system. The user-defined logging uses the familiar set of functions. Information is logged as a Python dict of key/value pairs. It is certainly possible to log English sentences, but breaking up the information into a more structured form will simplify querying for it later.

- write()
- log_error()
- log_warn()

- log_info()
- log_debug()
- log_trace()

7.1.5 Analysis

- check()
- dot_execution() - write an execution graph for the currently running program
- LogStatus
- find()
- query()
- Record

7.1.6 Data dependencies

- PREVIOUS - a syntax where the user can specify that the output of a previously executed task as input for future task.
  - handles dependencies between templates. PREVIOUS can be both implicit and explicit.
  - The semantics of PREVIOUS are affected by the template type.

---

class tigres.**InputTypes**(*name=None*, *list_=None*)
List of types for inputs of a Task.

Example

```
>>> from tigres import *
>>> task1_types = InputTypes('Task1Types', [int, str])
```

class tigres.**InputArray**(*name=None*, *values=None*)
Array of one or more InputValues, which will be inputs to a TaskArray in a Template.

Example

```
>>> from tigres import InputArray,InputValues
>>> input_array = InputArray("input_array1", [InputValues("values",[1,'hello world'])])
```

class tigres.**InputValues**(*name=None*, *list_=None*)
Values for inputs of a Task.

Example

```
>>> from tigres import InputValues
>>> values = InputValues("my_values", [1, 'hello world'])
```

tigres.**PREVIOUS**
PREVIOUS is a syntax where the user can specify that the output of a previously executed task as input for a task. PREVIOUS creates dependencies between templates and can be both implicit and explict.

•PREVIOUS: Use the entire output of the previous task as the input.

- `PREVIOUS.i`: sed to split outputs across parallel tasks from the previous task.It matches the i-th output of | the previous task/template to the i-th InputValues of the task to be run. This only works for parallel tasks.
- `PREVIOUS.i[n]`: Use the n-th output of the previous task or template as input.
- `PREVIOUS.taskOrTemplateName`: Use the entire output of the previous template/task with specified name.
- `PREVIOUS.taskOrTemplateName.i`: Used to split outputs from the specified task or template across parallel tasks. | Match the i-th output of the previous task/template to the i-th InputValues of the task to be run. This only works for parallel tasks.
- `PREVIOUS.taskOrTemplateName.i[n]` Use the n-th output of the previous task or template as input.

class `tigres.`**`TaskArray`**(*name=None*, *tasks=None*)
List of one or more Tasks, which will be executed in a Template

Instances act just like a list, except that they have a name.

class `tigres.`**`Task`**(*name*, *task_type*, *impl_name*, *input_types=None*, *env=None*)
Function or program. A task is the atomic unit of execution in Tigres.

Example

```
>>> def abide(dude): return dude - 1
>>> from tigres import Task, FUNCTION, EXECUTABLE
>>> fn_task = Task("abiding", FUNCTION, abide)
>>> exe_task = Task("more_abiding", EXECUTABLE, "abide.sh")
```

`tigres.`**`sequence`**(*name*, *task_array*, *input_array*, *env=None*, *redirect=False*)
List of tasks placed in series; If no inputs are given for any task, the output of the previous task or template is the input of the next.

Valid configurations of `task_array` and `input_array` are:

- **InputArray has zero or more InputValues and TaskArray has one or more tasks:** The task array will be run sequentially for each InputValues list given. There will be one sequential set run for each available core. The 2nd though nth task given will be fed the previous values from the task before. In the first implementation, InputValues must be provided.

Parameters

- **name** – an optional name can be assigned to the sequence
- **task_array** (*TaskArray or list*) – an array containing tasks to execute
- **input_array** (*InputArray or list*) – an array of input data for the specified tasks
- **env** (*dict*) – keyword arguments for the template task execution environment
- **redirect** (*bool*) – Turn on redirection. The standard output of the previous task is piped to standard input of the next task. IMPORTANT: PREVIOUS syntax does not work between tasks in this template.

Returns results from the last task executed in the sequence. If the results of the last sequence is from a python function the output type will be defined by the type returned otherwise executable output will be a string.

Return type object or str

Example

```
>>> from tigres import *
>>> def adder(a, b): return a + b
>>> task1 = Task("Task 1", FUNCTION, adder)
>>> task2 = Task("Task 2", EXECUTABLE, "echo")
>>> tasks = TaskArray(None, [task1, task2])
>>> inputs = InputArray(None, [InputValues(None, [2, 3]), InputValues(None, ['world'])])
>>> sequence("My Sequence", tasks, inputs)
'world'
```

tigres.**merge**(*name*, *task_array*, *input_array*, *merge_task*, *merge_input_values=None*, *env=None*)

Collect output of parallel tasks

Single 'merge' task is being fed inputs from a set of parallel tasks. The parallel task inputs (`input_array`) must be explicitly defined with `PREVIOUS` or explicit values.

Note: If the following conditions are met: `a)` no inputs are given to the parallel task (`input_array`) `b)` the results of the previous template or task is iterable `c)` there is only one parallel task in the `task_array` then each item of the iterable results becomes a parallel task.

Parameters

- **name** – The name of the merge
- **task_array** (*TaskArray*) – array of tasks to be run in parallel before last task
- **input_array** (*InputArray or list or None*) – array of input values for task array
- **merge_task** (*Task*) – last task to be run after all tasks on task array finish. This task will receive a *list* of values, whose length is equal to the length of the task array or the list .
- **merge_input_values** (*InputValues or None*) – List of input values for last task, or the outputs of each task (as a list), if None.
- **env** – keyword arguments for the template task execution environment

Returns results from the merge task. If the results of the merge task is from a python function the output type will be defined by the type returned otherwise executable output will be a string.

Return type object or string

tigres.**split**(*name*, *split_task*, *split_input_values*, *task_array*, *input_array*, *env=None*)

Single 'split' task feeding inputs to a set of parallel tasks. The parallel task inputs (`input_array`) must be explicitly defined with `PREVIOUS` or explicit values.

Note: If the following conditions are met: `a)` no inputs are given to the parallel task (`input_array`) `b)` the results of the `split_task` is iterable `c)` there is only one parallel task in the `task_array` then each item in the iteration becomes a parallel task.

Parameters

- **name** (*str*) – Name of the split
- **split_task** (*Task*) – first task to be run, followed by tasks on task_array
- **split_input_values** (*InputValues or list*) – input values for first task
- **task_array** (*TaskArray*) – array of tasks to be run in parallel after first
- **input_array** (*InputArray or list*) – array of input values for task array, default PREVIOUS
- **env** – keyword arguments for the template task execution environment

Returns Output of the parallel operation

Return type list(object)

tigres.**parallel**(*name*, *task_array*, *input_array*, *env=None*)

List of tasks processing their inputs in parallel.

Valid configurations of `task_array` and `input_array` are:

1. **TaskArray and InputArray are of equal length:** Each `InputValues` in the `InputArray` will be match with the `Task` at the same index in the `TaskArray`
2. **InputArray has zero or one InputValues and TaskArray has one or more tasks:** If there is one `InputValues`, it will be reused for all tasks in the `TaskArray`. If there are no `InputValues` then the output from the previous task or template will be the input for each Task in the TaskArray.
3. **TaskArray has one task and InputArray has one or more InputValues:** The one `Task` will be executed with each `InputValues` in the `InputArray`

Note: If the following conditions are met: `a)` no inputs are given to the parallel task (`input_array`) `b)` the results of the previous template or task is iterable `c)` there is only one parallel task in the `task_array` then each item of the iterable results becomes a parallel task.

Parameters

- **name** (*str or None*) – an optional name can be assigned to the sequence
- **task_array** (*TaskArray*) – an array containing tasks to execute
- **input_array** (*InputArray*) – an array of input data for the specified tasks
- **env** – keyword arguments for the template task execution environment

Returns list of task outputs

Return type list

Example

```
>>> from tigres import *
>>> def adder(a, b): return a + b
>>> task1 = Task(None, FUNCTION, adder)
>>> task2 = Task(None, EXECUTABLE, "/bin/echo")
>>> inputs = InputArray(None,[InputValues("One",[1,2]), InputValues("Two",['world'])])
>>> tasks = TaskArray(None,[task1, task2])
>>> parallel("My Parallel", tasks, inputs)
[3, 'world']
```

tigres.**parallel_sequential**(*name*, *task_array*, *input_array*, *env=None*, *redirect=False*)

A list of tasks is given in the task_array. Those tasks will be run sequentially multiple times in parallel lanes depending on the size of input_array.

Valid configurations of `task_array` and `input_array` are:

- **InputArray** has zero or more InputValues and TaskArray has one or more tasks: If the InputArray has zero entries, then parallel_sequential is the same as sequential with no inputs. [PROVISIONAL] Otherwise, input_array will be formatted as a rectangular matrix (in python a list of lists) where each row is the input list for a sequence in a lane. All subsequent tasks in the sequence will use implicit PREVIOUS. This means that any parameters needed in later sequence steps will need to passed in at the top and passed down. This requirement might be changed in later versions.

Parameters

- **name** – an optional name can be assigned to the sequence
- **task_array** (*TaskArray or list*) – an array containing tasks to execute
- **input_array** (*InputArray or list*) – an array of input data for the specified tasks
- **env** (*dict*) – keyword arguments for the template task execution environment
- **redirect** (*bool*) – Turn on redirection. The standard output of the previous task in a sequence is piped to standard input of the next sequence task. IMPORTANT: PREVIOUS syntax does not work between sequence tasks in this template.

Returns results from the last task executed in the sequence. If the results of the last sequence is from a python function the output type will be defined by the type returned otherwise executable output will be a string.

Return type object or str

tigres.**log_debug**(**args*, ***kwargs*)
Write a user log entry at level DEBUG.

This simply calls *write()* with the *level* argument set to DEBUG. See documentation of *write()* for details.

tigres.**log_error**(**args*, ***kwargs*)
Write a user log entry at level ERROR.

This simply calls *write()* with the *level* argument set to ERROR. See documentation of *write()* for details.

tigres.**log_trace**(**args*, ***kwargs*)
Write a user log entry at level TRACE.

This simply calls *write()* with the *level* argument set to TRACE. See documentation of *write()* for details.

tigres.**log_debug**(**args*, ***kwargs*)
Write a user log entry at level DEBUG.

This simply calls *write()* with the *level* argument set to DEBUG. See documentation of *write()* for details.

tigres.**log_info**(**args*, ***kwargs*)
Write a user log entry at level INFO.

This simply calls *write()* with the *level* argument set to INFO. See documentation of *write()* for details.

tigres.**log_warn**(**args*, ***kwargs*)
Write a user log entry at level ERROR.

This simply calls *write()* with the *level* argument set to ERROR. See documentation of *write()* for details.

tigres.**set_log_level**(*level*)
Set the level of logging for the user-generated logs.

After this call, all messages at a numerically higher level (e.g. DEBUG if level is INFO) will be skipped.

Parameters **level** (*int*) – One of the constants defined in this module: NONE, FATAL, ERROR, WARN[ING], INFO, DEBUG, TRACE

Returns None

Raise ValueError if *level* is out of range or not an integer

tigres.**start**(*name=None*, *log_dest=None*, *execution='tigres.core.execution.plugin.local.ExecutionPluginLocalThread'*, *recover=False*, *recovery_log=False*, ***kwargs*)
Configures and initializes tigres monitoring

WARNING: this will end any previously started programs

Parameters

- **name** –
- **log_dest** –
- **name** – Name of the program (Default: auto-generated name)
- **log_dest** – The log file to write to. (Default: auto-generated filename)
- **execution** – The execution plugin to use
- **recover** – True if you would like to run the program in recovery mode
- **recovery_log** – path to the recovery log file
- **kwargs** – (log_meta) dictionary containing extra metatdata to log for this workflow (log_*) Additional keywords to pass to `monitoring.log.init()`, with the *log_* prefix stripped

Returns Workflow identifier

Return type str

Raises TigresException

`tigres.`**`end`**`()`

Clear the Tigres monitoring.

return None

`tigres.`**`write`**`(`*level*, *activity*, *message=None*, ***kwd*`)`

Write a user log entry.

If the API is not initialized, this does nothing.

Parameters

- **level** – Minimum level of logging at which to show this
- **activity** – What you are doing
- **message** – Optional message string, to enable simple message-style logging
- **kwd** – Contents of log entry. Note that timestamp is automatically added. Do not start keys with *tg_*, which is reserved for Tigres, unless you know what you are doing.

Returns None

Example:

```
# Write a simple message
write(Level.WARN, "looking_up", "I see a Vogon Constructor Fleet")
# Or, using the more convenient form
warn("looking_up", "I see a Vogon Constructor Fleet")
# Use key/value pairs
warn("looking_up", vehicle="spaceship", make="Vogon", model="Constructor", count=42)
```

`tigres.`**`check`**`(`*nodetype*, ***kwargs*`)`

Get status of a task or template (etc.).

Parameters

- **nodetype** (str, See `NodeType` for defined values) – Type of thing you are looking for
- **state** (*str*) – Only look at nodes in the given state (default=DONE, None for any)

- **multiple** (*bool*) – If True, return all matching items; otherwise group by nodetype and, if given, name. Return only the last record for each grouping.
- **names** (*list of str, or str*) – List of names (may be regular expressions) of nodes to look for.
- **program_id** – Only checks the nodes for the specifed program
- **template_id** – only checks the nodes that belong to the template with the specified template_id.

Returns list(LogStatus). See parameter *multiple*.

tigres.**find**(*name=None*, *states=None*, *activity=None*, *template_id=None*, *task_id=None*, ***key_value_pairs*)

Find and return log records based on simple criteria of the name of the activity and matching attributes (key/value pairs).

Parameters

- **name** – Name of node, may be empty
- **states** (*list of str or None*) – List of user activities and/or Tigres states to snippets, None being "all".
- **activity** (*str*) – One specific user activity (may also be listed in 'states')
- **template_id** (*str*) – Optional template identifier to filter against
- **task_id** (*str*) – Optional task identifier to filter against
- **key_value_pairs** – Key-value pairs to match against. The value may be a number or a string. Exact match is performed. If a key is present in the record, the value must match. If a key is not present in a record, then it is ignored.

Returns List of *Record* instances representing the original user log data

tigres.**query**(*spec=None*, *fields=None*, *metadata=False*)

Find and return log records based on a list of filter expressions.

Supported operators for *spec* are:

•>= : Compare two numeric values, A and B, and return whether A >= B.

•= : Compare two numeric or string values, A and B, and return whether A = B.

•<= : Compare two numeric values, A and B, and return whether A <= B.

•> : Compare two numeric values, A and B, and return whether A > B.

•!= : Compare two numeric or string values, A and B, and return whether A != B.

•< : Compare two numeric values, A and B, and return whether A < B.

•~ : Compare the string value, A, with the regular expression B, and return whether B matches A.

Example expressions *foo > 1.5* – find records where field foo is greater than 1.5, ignoring records where foo is not a number.

foo ~ 1.d – find records where field foo is a '1' followed by a decimal point followed by some other digit.

Parameters

- **spec** (*list of str*) – Query specification, which is a list of expressions of the form "name op value". See documentation for details.
- **fields** (*None or list of str*) – The fields to return, in matched records. If empty, return all

- **metadata** – set this to True if you would like meta data for the program to be returned

Returns Generator function. Each item represents one record.

Return type list of Record (metadata = False), list of tuple (Record,metadata)

Raise BuildQueryError, ValueError

tigres.**start**(*name=None*, *log_dest=None*, *execution='tigres.core.execution.plugin.local.ExecutionPluginLocalThread'*, *recover=False*, *recovery_log=False*, ***kwargs*)

Configures and initializes tigres monitoring

WARNING: this will end any previously started programs

Parameters

- **name** –
- **log_dest** –
- **name** – Name of the program (Default: auto-generated name)
- **log_dest** – The log file to write to. (Default: auto-generated filename)
- **execution** – The execution plugin to use
- **recover** – True if you would like to run the program in recovery mode
- **recovery_log** – path to the recovery log file
- **kwargs** – (log_meta) dictionary containing extra metatdata to log for this workflow (log_*) Additional keywords to pass to `monitoring.log.init()`, with the *log_* prefix stripped

Returns Workflow identifier

Return type str

Raises TigresException

tigres.**end**()

Clear the Tigres monitoring.

return None

class tigres.**Level**

Defined levels, and some related functions.

•NONE (0): Nothing; no logging

•FATAL (10): Fatal errors

•ERROR (20): Non-fatal errors

•WARNING (30): Warnings, i.e. potential errors

•INFO (40): Informational messages

•DEBUG (50): Debugging messages

•TRACE (60): More detailed debugging messages

•Other levels up to 100 are allowed

Note that the numeric equivalents of these levels is not the same as the Python logging package. Basically, they are in the reverse order – higher levels are less fatal. The *Level* class has static methods for manipulating levels and converting to/from the Python logging levels.

static names()
Return list of all names of levels, in no particular order.

Returns List of level names

Return type list of str

static to_logging(*level*)
Get Python logging module level.

Parameters **level** – Tigres logging level

Returns Equivalent Python logging module level.

Return type int

static to_name(*levelno*)
Convert a numeric level to its string equivalent.

Parameters **levelno** (*int*) – Numeric level

Returns String level, or empty string

Return type str

static to_number(*level_str*)
Convert a string level to its numeric equivalent.

Parameters **level_str** (*str or basestring*) – String level (case-insensitive)

Returns Numeric level, or NONE

Return type int

tigres.**get_results**()
Get the results from the latest task or template of the running Tigres program

Returns

class tigres.**LogStatus**(*rec*)
Provide convenient access to status information related to a single log entry.

Instances of this class are returned by the *check()* function.

to_json()
Return JSON representation of status.

class tigres.**Record**(*rec=None*)
Record used for formatting

from_dict(*rec*)
Add values from dict

Convert timestamp to a number of seconds since the epoch (1-1-1970) Convert the level name to a number.

intersect_equal(*other*, *ignore=()*)
Check whether all keys/values in given record, which are present in this one, are the same.

Parameters

- **other** (*Record*) – The record to compare to
- **ignore** (*list of str*) – Ignore these keys

Return type bool

project(*fields*, *copy=True*)
Project the record onto just these fields.

Parameters

- **fields** (*list of object*) – The fields to project onto
- **copy** (*bool*) – Copy to a new record, or in-place

Returns New record, or nothing if copy=False

Return type Record or None

`tigres.`**`dot_execution`**(*path='.'*)

Creates an execution graph of the currently running Tigres program and writes it to a file

Parameters **path** – the file path to write the DOT file to

Returns

Sideeffect writes a DOT file of the currently running Tigres `tigres.core.monitoring.state.Program`

7.2 *tigres.utils*

platform Unix, Mac

synopsis Tigres shared classes/methods/constants

module author Gilberto Pastorello <gzpastorello@lbl.gov>, Val Hendrix <vchendrix@lbl.gov>

7.2.1 Functions

- `get_new_output_file()` - Get a new unique output file name.

7.2.2 Classes

- `Execution` - Execution plugins available for a Tigres program
- `State` - States of a Tigres program
- `TigresException` - A Tigres Program Exception
- `TaskFailure` - Object to be returned in case of failures in task execution

class `tigres.utils.`**`Execution`**

Execution plugins available

•LOCAL_THREAD

•LOCAL_PROCESS

•DISTRIBUTE_PROCESS

•SGE

•SLURM

classmethod **`get`**(*name*)

get the execution plugin for the name

Example

```
>>> from tigres.utils import Execution
>>> Execution.get('EXECUTION_LOCAL_THREAD')
'tigres.core.execution.plugin.local.ExecutionPluginLocalThread'
```

Parameters **name** – execution plugin string

Returns plugin module

Return type str

class tigres.utils.**State**
Tigres State constants

- NEW - NEW, CREATED or START
- READY - WAIT, WAITING or READY
- RUN - RUN, RUNNING or ACTIVE
- DONE - DONE, TERMINATED or FINISHED
- FAIL - FAIL or FAILED execution (e.g., from exception)
- UNKNOWN - UNKNOWN

classmethod paste(*name*, *state*)
Combine a name and a state into a single string.

class tigres.utils.**TaskFailure**(*name=''*, *error=TigresException('Task execution failed',)*)
Object to be returned in case of failures in task execution

Parameters

- **name** (*str*) – Name/description for error
- **error** (*Exception*) – Exception that occurred, if available

exception tigres.utils.**TigresException**
A Tigres Program Exception

tigres.utils.**get_new_output_file**(*basename='tigres'*, *extension='log'*)
Get a new unique output file name.

Parameters

- **basename** (*str*) – Prefix for name
- **extension** (*str*) – File suffix (placed after a '.')

Returns Name of file (caller should open)

CHAPTER

EIGHT

NEW FEATURE EXAMPLES

The following are example workflows for new Tigres features.

- **Sequence Template Enhancement: Task output redirection**
  - input: `data.csv`
  - `example_sequence_redirect.py`
- **Parallel Sequential Template**
  - `example_parallel_sequential.py`

CHAPTER

NINE

CONTACT US

For general information see the Tigres home page.

- **Users' Support Mailing List:** tigres-support at lists.lbl.gov

CHAPTER

TEN

LICENSE AGREEMENT

Tigres is released under the license below and is subject to the following licensing terms:

CHAPTER

ELEVEN

COPYRIGHT NOTICE

CHAPTER

TWELVE

CHANGE LOG

12.1 v0.2.0

This is release contains Failure Recovery, new Parallel Sequential template, execution now supports elastic resources, execution optimizations and other minor enhancements. The following major issues related to this release are:

- #140: host_id for metadata is always localhost
- #139: Workflow Id returned from tigres.start()
- #138: Better logging metadata
- #137: Sequence template redirect integrated with Parallel Sequential Templates
- #136: File Exists Error with recovery tests
- #135: Sequence Template redirection
- #132: Task log name truncated upon log creation
- #131: "daemon" thread parameter misspelling
- #126: Create new template: parallel_sequential
- #125: Simple tool that mines logs for executions times
- #124: Optimize Thread, Process and Distribute parallel execution

12.2 v0.1.1

This is the First Usability Bug Fix (Alpha) Release of Tigres. It has the following major bug fixes can be found in Bitbucket.

- #075 incorrect dot_execution() output
- #076 cannot specify first-time tasks that take no input
- #077 distinguishing failed merge task from failed parallel task
- #087 multi-quote/nested text incorrectly parsed from log
- #094 tigres-log check –worklfow throws an error
- #097 Incorrect execution time for DISTRIBUTE_PROCESS tasks
- #098 Execution process template ends in ? state on success
- #099 'tigres-log check –name' not filtering by name

- #100 'tigres-log check' without –name attribute throws exception
- #101 'tigres-log check' should add node_type to output records
- #102 tigres-log check should only check the last workflow run
- #104 Execution Distribute does not utilize all cpus
- #105 tigres-log check - timestamp without milliseconds throws Exception
- #106 Task Width defaults to virtual cpu count and not physical
- #111 Execution when pickling distribute task give unhelpful error message Execution Model
- #113 testing SLURM with patcher not returning results
- #120 Update HPC tutorial documentation for SLURM resources
- #121 setup.py is skipping installation of cloud for python 2.7.x
- #122 Clean up error handling in tigres.core.execution.utils.TaskClient

12.3 v0.1.0

This is the First Usability (Alpha) Release of Tigres. It has a Python API with the following feature support.

- **Template Support:**
  - sequence, split, parallel, merge
- **Monitoring Support**
  - system event logging
  - user event logging with ability to set logging levels
  - log analysis support with interfaces for checking status, simple find and query syntax for filtering log records.
  - generate DOT graph file from currently running program.
- **Execution:**
  - Support for desktop, cluster and HPC execution

t