

# Frontier® Rapids User Guide

## Introduction

Frontier® Rapids is an environment for running native applications on the Frontier Grid Computing Platform. By native applications, we mean applications that were not originally developed to run on the Frontier Platform. These applications can be 3<sup>rd</sup> party applications (e.g., statistical or bioinformatics programs) or proprietary applications. In general, native applications are applications developed and compiled to run on specific operating system platforms (e.g., Windows or Linux), but native applications can also be applications developed in VM-based and scripting languages. Frontier Rapids allows any of these native application types to run on the Frontier Grid Platform without modification.

Frontier Rapids provides the ability to run a large number of instances of an unmodified native application as a single distributed job on Frontier. For many classes of applications, this can result in dramatically faster execution times. Using Frontier Rapids with existing applications can enable you to solve problems that could not be solved when running those applications in conventional computing environments.

Setting up a Rapids job is simple. A Rapids job consists of one or more native tasks where each native task is an independent invocation of a platform-specific executable. The native tasks in a job can run with identical or task-specific input data. The inputs can be command-line parameters, data files, or both. Native tasks produce outputs that are returned to the Rapids application. The outputs consist of one or more data files for each task. The Frontier scheduler ensures that native tasks get scheduled on the appropriate platforms (i.e., nodes able to execute the native code). If the native executable is available for multiple platforms, then the native job can be configured to include the multiple executables. When a node becomes available, Frontier will automatically send it the appropriate executable for its platform type.

Rapids can manage jobs that run a large class of executables as native tasks, namely, any executable that can:

- run without a graphical user interface;
- read input from the command line and/or from files in the current directory (or a subdirectory of the current directory); and
- write output to standard output and/or files in the current directory (or a subdirectory of the current directory).

It is possible to run executables that don't meet all these conditions, but it requires special setup that is beyond the scope of this document.

This document assumes a basic familiarity with the Frontier Grid Computing Platform. The Foundation section of the Frontier API Whitepaper provides the required background. This document is available from:

<http://www.parabon.com/dev-center/sdk/api/index.html>

# Installation

## ***Installation Requirements***

There is a Rapids installer for both Windows (including Windows 2000, Windows XP, Windows Server 2003, and Windows Server 2008) and Generic Unix (includes UNIX, Linux, and MAC OS X). The host must also have an installation of the Java Runtime Environment (JRE) 5.0 or later.

To run Rapids jobs, you will also need a Frontier client account with credentials installed on your machine. You can do this by signing up for an account, downloading the Frontier SDK, and registering your account with the Frontier Server. When you register your account with the Frontier server, be sure to remember the keystore password you use as it will be needed later.

Since you will be running native code, some special Frontier setup is required. Specifically,

- Your client account requires privilege to run native code.
- Your client account must be associated with a node group containing nodes that are enabled to run native code.

Send e-mail to [support@parabon.com](mailto:support@parabon.com) for help in setting up your account or setting up your node group.

## ***Installation and First Use***

To install Rapids, you need only run the appropriate installer and follow the prompts. There is both a Windows and a generic UNIX installer.

The first time you run Frontier Rapids, your installation will be configured. You will be asked which Frontier server you are using and for the keystore password you established when you set up your Frontier account. The configuration will also install the sample Rapids templates (see below).

If you need to subsequently change the Rapids configuration, you can invoke `rapids` with the `Q` or `C` option (`/Q` or `/C` on Windows; `-q` or `-c` on UNIX/Linux platforms). The `Q` option does a quick configuration. The `C` option enables advanced configuration.

## **System Overview**

Frontier Rapids consists of five major components: The Rapids Application, the Rapids Template Repository, the Rapids Job Repository, the Native Wrapper Application, and the Native Wrapper Task. These components are described below.

### ***Rapids Application***

The Rapids Application is a command-based application that supports the creation, execution, monitoring, and control of native jobs on Frontier. Jobs are created from templates that define the common executables and data elements that comprise the job tasks as well as the data sets that are used to create instances of the job.

## ***Rapids Application Program Interface***

The Rapids API is a Java API that provides all the capabilities of the Rapids application. The Rapids API is used when there is a need to create, launch, monitor, and control a Rapids job from a Java application.

## ***Rapids Template Repository***

The Rapids Template Repository is a Repository of templates that can be used to create jobs from within the Rapids application. The first step in running a native application on Frontier is to develop a template for that application. This is typically done by starting with one of the Rapids sample templates and modifying the sample to meet your needs.

## ***Rapids Job Repository***

The Rapids Job Repository is a repository of information about job instances. The repository contains all the local information about the state of the job. Normally users don't directly access the Job Repository. It is only accessed through the Rapids application.

## ***Native Wrapper Application***

The Native Wrapper Application is a Frontier application that launches Rapids jobs on Frontier, monitors the job execution, and keeps the local state of the job (stored in the Job Repository) up to date. When you launch a Frontier job using Rapids, an instance of the Native Wrapper Application is started to manage that job. This instance runs in background and does not require any direct user interaction.

## ***Native Wrapper Task***

The Native Wrapper Task is a generic Frontier task that manages the execution of native code on a Frontier node. On Frontier, each native task has its own temporary directory on the Frontier compute node. The Native Wrapper Task is passed all the native task information (e.g., executables, parameters, data files) to run its particular task instance. It then:

- writes the task executables to the task temporary directory
- writes the task input files to the task temporary directory
- launches the executable with the appropriate task parameters
- monitors execution and sends back task status information
- periodically sends back the interim state of task result files
- sends back the final versions of the output files when the executable exits
- if an error occurs, error information is returned that can be used to debug the problem

Normally, end users need only concern themselves with the Rapids application and the construction of job templates. These two areas will be the focus of the remainder of this document.

## **Rapids Application**

The Rapids application supports the creation, execution, monitoring, and control of native jobs created from job templates. Users can create multiple jobs that can run on Frontier simultaneously.

The Rapids application is started from an appropriate launch script (rapids.sh on UNIX/Linux and rapids.bat on Windows). The application can be used in interactive mode or command mode. If launched without an argument, the application is in interactive mode. For example,

```
$ rapids.sh
rapids> newjob mult mult1 mult1
rapids> launch 0
rapids> quit
```

If a command is provided as an argument to the Rapids application, the command is immediately executed and Rapids exits when the command is complete. For example:

```
$ rapids.sh newjob mult mult1 mult1
$ rapids.sh launch 0
```

The form you use is a largely a matter of personal taste.

The Rapids application currently supports the following commands:

newjob	- create a new job
launch	- launch a job and start its listener
listen	- restart the listener for a job
stop	- stop a job listener
kill	- kill a job or task
status	- return status information for a job
delete	- remove the state information for a completed job
remove	- remove a job from the Frontier Server (advanced)
log	- display a job or task log
list	- list jobs
help	- display a list of available commands
upload	- uploads a client scope element to the server
update	- adds new tasks to an existing job
quit	- exit Rapids

Additional information on these commands is available using the rapids help command (the help files are in the docs/commands directory).

## Rapids Template Structure

Rapids templates define Frontier native jobs and provide a location for the placement of job results. The function of a template is to provide the Rapids application with all the information required to run a native job on Frontier. Rapids Templates are created in the templates directory. By default, the templates directory is created in Rapids home directory. Under Windows, the Rapids home directory is created in the user's profile folder. Under UNIX/Linux, the Rapids home directory is created in the user's home directory. Each subdirectory in the templates directory is a job template. The name of the directory is the template name. A template directory may contain any or all of the following:

## ***common\_elements***

This directory contains data files that are common to all tasks in the job. These can be non-native elements or native elements.

A non-native element is a single file that is sent to all instances of a task regardless of the platform that task is running on. Non-native elements are represented as simple files in the `common_elements` directory.

A native element is a single logical file that contains different versions for different platforms (e.g., an executable). Native elements are represented in the `common_elements` directory as a directory. The name of the directory is the name that will be used for the element remotely. The directory contains one sub-directory for each supported platform (e.g., WIN, UNIX/Linux, MACOS). The platform subdirectory contains the version of the element for that specific platform. The file in the platform subdirectory may or may not have the same name as the native element directory name. When Rapids copies this file to the compute node, it will be given the name of the native element directory name.

There are two common elements that are required: `task.properties` and `startup.cmd` (these should be native elements rather than non-native elements):

The `task.properties` file contains the following properties:

```
SHELL           - the name of the shell to use to run the executable
SHELL_OPTIONS  - options to pass to the shell
COMMAND_LINE  - the command line that is used to launch the
                  executable
```

These tell the task which executable to run. Normally, a `task.properties` file is created once for a given platform and can be reused after that (e.g., the `task.properties` files in the samples can probably be used in your application without being altered).

The `task.properties` file may also contain the following optional properties that govern task execution:

```
DISK_QUOTA     - limits the total amount of disk space that can be
                  used by the task (in bytes). If the task's disk
                  usage exceeds this amount, the task will be
                  terminated.
EXCEPTION_FILE - this is the file name for the exception file
                  to be generated by a native task. See Error
                  Handling under Advanced Topics below.
```

The `startup.cmd` element is a platform-specific script that is used to run your job. This script is usually tailored for a specific job. Typically, this is a simple script that sets up the environment for the job to run and then runs the main executable for the job. See the sample templates for example `startup.cmd` scripts.

You will generally want to include other common elements in your task. Typically, a main executable is passed as yet another native data element. Note that this is not required, since the

startup.cmd script may simply run an executable that is already known to be on the compute node (or do all the work itself). Also, if there are input files that are shared by all tasks, they will be passed as common\_elements. If the same file can be used across platforms, then these files should be passed as non-native elements. If the format of the input files is platform specific, then these files should be passed as native elements.

Every file in the common\_elements directory is considered an element. If you have certain meta files that you want rapids to ignore (e.g., CVS or .svn) you can add these to the ".rapids.exclude" file in the Rapids install directory. All files listed in the ".rapids.exclude" file are ignored when loading templates.

### ***inputFiles.txt***

In many cases, you want to pass different tasks different versions of an input file (i.e., so each can solve a different part of a problem). To do this, you specify the names of the input files in the inputFiles.txt file and then you supply the actual files in a dataset in the datasets directory (see datasets below).

### ***outputFiles.txt***

As mentioned above, a task can produce one or more output files. These files are written to the task's working directory on the compute node. It is common to want some or all of the output files to be returned to your local machine (so you can see the results). You specify that you want this to happen by including the relevant file names in the outputFiles.txt file. The resulting files are written to a result set in the resultsets directory (see below).

### ***client\_elements.txt***

Some jobs need to reference client scope elements (see Client Scope Elements below). The client\_elements.txt file contains one line per client scope element required by the job. This line is the name of a client scope element that appears in the top level templates/client\_elements directory. Client scope elements are described in detail under advanced topics.

### ***datasets***

The datasets directory contains input data sets for instances of the job defined by the template (i.e., each run of the job uses a different dataset). Each directory in the datasets directory is a separate dataset. The name of this directory is the dataset name. The files in this directory correspond to the input files defined for the job in the "inputFiles.txt" file. For each input file name appearing in the inputFiles.txt file, there is a set of actual input files in the dataset directory (one for each task in the job). The naming convention for these files is <input-file><task-index>.<ext>, where the extension is optional. For example, if the inputFiles.txt file contains "foo" and "bar.dat" and the job has 3 tasks, the input files will be named "foo0", "foo1", "foo2", "bar0.dat", "bar1.dat", "bar2.dat".

A datasets directory may also contain a parameters.txt file. If this file exists, it contains parameters that are to be passed to the task (one line per task). A task can have both input files and parameters.

## ***resultsets***

The resultsets directory is used to store job output. Each directory in the resultsets directory contains a single result set (i.e., the output files from a single run of the job). The result sets in the resultsets directory are created automatically when the job is run. The name of the directory is the name of the result set. When a job is run via the Rapids application the user specifies a result set name. Rapids will place the task output files (as defined in outputFiles.txt) into a result set directory with this name. For each output file named in the outputFiles.txt file, there will be a set of actual files in the result set directory (corresponding to the tasks in the job). The naming convention for these files is <output-file><task-index>.<ext>, where the extension is optional. For example, a 3 task job with an output file named "output.dat" will have three output files in its resultset directory called "output0.dat", "output1.dat", and "output2.dat".

For reference, a file called jobId.txt is written into the job's resultsets directory. This file contains the corresponding job's job ID.

## **Advanced Topics**

### ***Error Handling***

The return code from the task's command script is used to determine whether the task has succeeded or failed. If the script returns zero, the task is considered successful. If the script returns non-zero, the task is considered failed. Normally, a failed task will exit and the failure code/message will be written to the task log (which can be viewed with the Rapids log command). Such a task will not be rescheduled on another node.

Normally, if a task fails on one Frontier node, it can be expected to fail on all nodes (e.g., because of a problem in the task itself). There are however, cases where you might want a failed task to be rescheduled on another node (e.g., because of a local resource problem). This can be accomplished by specifying an EXCEPTION\_FILE for the task in the task's task.properties file. The native application should then write a file of this name to the task's working directory if the application encounters a node-specific error condition. If an exception file is specified and the task's command script exits with a non-zero value, the Rapids Wrapper task will look for the specified file and use its contents as the body of a runtime exception to be thrown by the task. Since the task will now exit with an exception, it will be automatically rescheduled on a different Frontier node where it may find the required resources. Note that if a task fails on too many nodes, Frontier will automatically abort the task. This prevents a scheduling loop for malformed tasks.

### ***Client Scope Elements***

Rapids supports the definition of client scope elements. Client scope elements are native or non-native elements that are associated with a client rather than a job. The fact that they are client scope means that they can be referenced from multiple client jobs. Furthermore, because their existence is not tied to a job, they persist on the server (and in compute node caches) even after jobs that use them are deleted. This can result in a very significant reduction in the amount of data that must be transmitted to the server and to compute nodes. Client scope elements are deleted from the server once they have not been referenced by a job for 30 days.

Client scope elements are defined in the `client_elements` directory in the top-level templates directory. Elements are created in this directory in exactly the same way as job scope elements are defined in the common elements directory of a job template. Client scope elements are named elements. The name of the element is the name of the corresponding element file (for non-native elements) or directory (for native elements) in the top-level `client_elements` directory. Before a client scope element can be referenced in a job definition, it must be uploaded to the server using the Rapids upload command. The upload command takes the name of the client element to be uploaded. Once a client scope element has been created and uploaded to the server, it can be referenced by jobs. Job templates indicate that a client scope element is to be included in the job by adding the name of the element in the `client_elements.txt` file in the template directory.

Client scope elements can be very useful for relatively heavyweight native applications. One way of using them is to archive an entire application installation directory and make that archive a client scope element. Archives can be created as a `tar/gzip` file on UNIX/Linux or as a self-extracting `zip/rar` file on Windows. The `startup.cmd` file can then `unzip/untar` the file into the application directory. Since the element is client scope, it generally only gets sent to the compute node once (rather than once per task or job).

### ***Interim Progress Reports***

The Rapids environment supports returning interim progress from tasks. An interim progress report is a Frontier task status report that includes the current state of all task output files. These results are saved to the JobRepository and the interim output files are saved to the output resultset. The report interval parameter in the Rapids configuration determines how frequently interim progress reports are returned. The default is every 5 minutes.

### ***Task Checkpointing***

The execution of a task on a Frontier node can be interrupted for a variety of reasons. Interrupted tasks can be handled in two ways, they can be restarted from the beginning, or they can be restarted from where they were when they were interrupted. In order to do the latter, the native executable needs to support checkpointing. In this context, checkpointing means that the executable writes its state out to the task directory periodically, and if restarted, it will read in that state and pick up where it left off. If your native executable supports checkpointing, you should set the checkpoint flag parameter in the Rapids configuration to true. If this flag is false, then your task will not checkpoint correctly. The default value of this parameter is false.

### ***Debugging***

The Frontier Service is designed to insulate the developer from problems with particular Engines (e.g., disk full). For this reason, when a task throws certain exceptions, the server will attempt to schedule the same task on a different engine. Sometimes this behavior can make debugging a job more difficult. To help with debugging, the Rapids configuration script includes a debug flag parameter which can be set to true or false. If set to true, all task exceptions are returned to the application. This setting should NOT be used for production runs, as it may cause tasks to fail due to transient Engine issues.

## Rapids Template Samples

The following samples are included in the Rapids distribution (see the templates directory).

### **generic**

The generic sample is a simple multi-platform sample. All this sample does is execute a script (called startup.cmd) that writes "Hello World!" into a file (called result.txt). The executable script is defined as a common native element with support for two platforms WIN and Linux. The Windows version is a cmd script and the Linux version is a shell script. There is a second common native element "task.properties" which was described above. As mentioned above, this sample creates a single output file. Since this output file appears in the outputFiles.txt file, it will be returned to the Rapids application and written to the specified datasets directory when the task completes.

This job doesn't include input parameters or data files so there are no datasets specified.

To launch a generic job, execute the following commands:

```
./rapids.sh
rapids> newjob generic 5 out1
    (note: this says create a 5 task job from the generic template,
    and save the result to the out1 result set; you can find this
    result set in templates/generic/resultsets/out1. This command will
    also output a job number).
rapids> launch x
    (note: x is the job number returned by the newjob command)
```

Once launched, you can track the status of the status of the job with the log and status commands.

### ***mult***

The mult sample is similar to the generic sample, but extends it in several important ways. The mult task reads one number from a file (called input.txt), a second number from the command line, multiplies the two together, and stores the result in a file (called result.txt).

This sample has three multi-platform common native elements. The task.properties is the same as above, the startup.cmd element is similar to the generic sample except that, instead of doing the work itself, it launches a separate executable (called mult). The mult executable is the third common native element.

Since the tasks in this sample read in a data file, the template includes an inputFiles.txt file. This also means that to run this job, you need to specify a dataset. Two sample datasets (called mult1 and mult2) are included in the distribution. Since this sample also includes command line parameters, the datasets also include a parameters.txt file which contains the input parameters.

Similar to the generic example, this example task produces a single output file. Since this output file appears in the outputFiles.txt file, it will be returned to the Rapids application and written to the specified datasets directory when the task completes.

To launch a mult job, execute the following commands:

```
./rapids.sh
rapids> newjob mult mult1 out1
    (note: this says create a job from the mult template, using the
    data set named mult1, and save the result to the out1 result
    set).
rapids> launch x
    (note: x is the job number returned by the newjob command. This
    launches the job on Frontier.)
```

Once launched, you can track the status of the job with the log and status commands.

### ***mult-cs***

This sample is the same as mult, except that it uses a client scope element for the mult executables. This means that the templates common\_elements directory doesn't contain the mult executables. Instead, the executables are located in the top level templates/client\_elements directory. The template directory contains the client\_elements.txt file which contains a reference to the mult client scope element.

Before the mult-cs job can be launched, the mult client scope element must be uploaded to the server. This is done as follows:

```
./rapids.sh
rapids> upload mult
```

Not this only needs to be done once, not each time the job is run.

The job itself is run exactly as described above.

### ***param***

This sample demonstrates tasks that only take their input from the command line. The task executable (square) takes a single parameter, squares it, and writes it to an output file called result. There is one dataset (param1) which contains a parameters.txt with one line for each task in the job. Each line is the single parameter that is passed to a task in the job.

To launch a param job, execute the following commands:

```
./rapids.sh
rapids> newjob param param1 out1
    (note: this says create a job from the param template, using the
    data set named param1, and save the result to the out1 result
    set).
rapids> launch x
    (note: x is the job number returned by the newjob command)
```

## ***Developing and Testing New Template***

Before attempting to develop your own templates, you should run the samples. This is the best way to become familiar with Rapids and to determine whether your cluster is configured correctly.

Once you have successfully run the samples, you can create a template for native jobs you want to run on Frontier. It is recommended that you start with a sample template (e.g., mult) and adapt it to your needs. When debugging your native job, be sure to use the log command to look at the job and task logs. If something goes wrong, these will usually give you enough information to debug your problem.

If you need assistance, please contact [support@parabon.com](mailto:support@parabon.com).