



Paraloop 1.2: The documentation

Introduction

The program `paraloop`, written in object `perl`, distributes the treatments on the processors of a parallel computer, hiding to the end user the architecture of the machine.

License

`paraloop` is governed by the CeCILL license, release 2. See <http://www.cecill.info> for the details.

General information:

Paraloop may be used when the work to be done may be easily splitted in N tasks. Each task reads the input data from some file, and each tasks writes the output data in a separate file, thus avoiding any synchronization problem. Each task is in fact a “loop”, the processing executed for each iteration is called the “atomic job”. Each task logs messages in a separate file.

Paraloop may be used on SMP multiprocessor machines, or on clusters. If a queueing system is intalled on the machine, Paraloop may use it (currently only PBS is supported), if there is no queuing system `paraloop` tries to optimize the processors used.

The object architecture makes easy to perform several different tasks with `paraloop`: you may use one of the general plugins (like Shell or BioPerl), or a more specific plugin (like Blast), or you may easily write a new plugin.

Installing and configuring paraloop

There is no installation script (sorry). The first step is to download the tar gz file, and to extract the files and directories:

```
ls -l
total 24
drwxrwxr-x  2 manu  prodom  4096 May 27 13:47 bin
drwxrwxr-x  2 manu  prodom  4096 May 27 13:47 documentation
drwxrwxr-x  2 manu  prodom  4096 May 27 13:47 etc
drwxrwxr-x  3 manu  prodom  4096 May 27 13:47 lib
```

Needed modules

There is no mandatory module to use `paraloop` in a general way, however it is better installing two `perl` products:

1. The `PbInput` object, which is extended by the Blast, the Bioperl and Dummy plugins, needs `bioperl`¹ to read and write the input files. So, if you want to use one of these plugins, you'll have to install `bioperl`.

¹ www.bioperl.org

- you may have more precise log files if you install the module `Time::HiRes`, as this let you retrieve timer information with a resolution better than 1 s. You may find this module from <http://search.cpan.org/~jhi/Time-HiRes-1.66/HiRes.pm>

Choosing a root directory

The directory in which `paraloop` is installed (the root directory) should be mounted on every processor you are planning to use: this is particularly true if you are working on a cluster. Moreover, the path to this directory should be the same on every cluster node. If you are an user, the easiest is to choose your home directory as root directory. If you are an administrator, a better solution is to choose `/usr/local`, `/usr/share`, etc. After extraction, you have the following directories:

```
ls -l
total 40
drwxrwsr-x  5 root root  4096 Apr 11 18:28 paraloop-1.2

# ls -l
total 1
drwxrwxr-x  2 root root   80 Nov 22 09:16 bin
drwxrwxr-x  2 root root  824 Nov 22 09:16 documentation
drwxrwxr-x  2 root root  120 Nov 22 09:16 etc
drwxrwxr-x  3 root root  200 Nov 22 09:16 lib
```

Defining the environment variable `$PARALOOP`:

This variable should be defined in order for `paraloop` to find the libraries. It should be set to the complete path of the `paraloop` directory:

```
setenv PARALOOP /usr/local/paraloop-1.1
```

You should set this environment variable in your `~/.csh` file as an user, or in `/etc/csh.cshrc` as an administrator. Or in the corresponding profile files for `sh`.

Calling `paraloop`:

You have several solutions to call the program:

- Using the whole path:
`$PARALOOP/bin/paraloop.pl`
- Defining an alias:
`alias paraloop $PARALOOP/bin/paraloop.pl`
- Creating a link from a directory already in the path. For an admin:
`cd /usr/local/bin;`
`ln -s $PARALOOP/bin/paraloop.pl .`
- Copying the file `paraloop.pl` to `/usr/local/bin`

From now on, we consider that `paraloop` may be called with the command:
`paraloop.pl`

configuring `paraloop`:

IMPORTANT: Several template configuration files, corresponding to several use cases, may be found in the directory `$PARALOOP/documentation/templates`

Many parameters must be specified for `paraloop` to work correctly. Some of them can be specified though the command line, but others must be specified in some configuration file. You should edit the following files:

- `$PARALOOOP/etc/paralooop.root.cfg`
- `$PARALOOOP/etc/paralooop.cfg`

This is particularly important if you are installing `paralooop` as an administrator, to be used by every user in the system.

The parameters you should set in those files are essentially the same, but their meaning is quite different:

- If some parameter is set in `paralooop.root.cfg`, its value will **not be changed** by any user. It is the place to choose the scheduler, for example, or any information relative to the whole site, its architecture, its policy.
- If some parameter is set in `paralooop.cfg`, it is rather considered as a default value. The users may override it, if needed.

The paralooop user configuration:

Every user may set some parameters for her personal use of `paralooop`. Those parameters may be specified in several locations:

- `~/.paraloooprc` This file will be always read by `paralooop`, if it exists, so you should put here the parameters you want to keep always to the same value. For instance, you may prefer to use `error` as a directory name for your error files (the default is `PARALOOOP_error`): in this case, you should write the following line in `~/.paraloooprc`:
`PARALOOOP_error_directory = error`
- Other configuration files: You may specify any configuration file with the `--cfg` switch. This can be convenient places to put parameters which are specific to some project.

The list of configurable parameters may be displayed with the command:
`paralooop.pl --parameters`

The order of the parameter files

The parameters are read from the command line or from the configuration files with the following priority. It is supposed here that `paralooop` was called with the switch

`--cfg f1.cfg,f2.cfg`

1. The switch on the command line
2. `$PARALOOOP/etc/paralooop.root.cfg`
3. `f1.cfg`
4. `f2.cfg`
5. `$HOME/.paraloooprc`
6. `$PARALOOOP/etc/paralooop.cfg`

When a parameter is set, its value is never modified by any other file. So the switches of the command line have the highest priority (but not all parameters can be set through the command line); the parameters set in the `etc/paralooop.root.cfg` file cannot be reset afterwards, etc. The file `etc/paralooop.cfg` defines default values.

Using paralooop

The input data:

We'll suppose in the following that your data have the following characteristics:

- They consist of a list of records
- Each record may be treated independently from the others.

A good example is a multifasta files on which each sequence must be treated one after another.

Specifying the Plugin to use:

In the following, we shall suppose you want to execute a blast for every sequence found in the fasta file, using the database `database/uniprot.fasta`, putting the result to some files starting with `BlastResult` in directory output. You'll use 10 processors for this purpose:

```
paralooop.pl --cfg blast.cfg --program Blast \  
  --db databases/uniprot.fasta \  
  --input input/seq.fasta --output output/BlastResult \  
  --ncpus 10
```

The `--program Blast` is a plugin specification, telling `paralooop` you want to use the plugin called `Blast`.

Specifying start, end:

If you want to perform the blast only from record nb. 0 to record nb. 1000 (included), you can specify the `--start` and `--end` switches.

The interleaved mode:

You may distribute the jobs in two modes:

- *slice mode*: sequences 0 to 99 are given to processor 1, seq 100 to 199 are given to processor 2, etc. This is the default mode.
- *Interleaved mode*: sequence 1 to cpu 1, sequence 2 to cpu 2, etc. This is useful if your data are ordered so that the first records have a longer atomic treatment than the last ones, etc. This mode is selected with the `--interleaved` switch.

Specifying files and directories:

Some switches or parameters let you specify the name of :

- the error directory (default `PARALOOOP_error`)
- the lock directory (default `PARALOOOP_lock`)
- the input file (*no default*)
- the database file (for some plugins only, *no default*)
- the output directory (*no default*)

You can use some special characters in specifying those parameters, which will be replaced at run time. The legal characters are described here:

<code>%h</code>	The hour part of time (11 for 11:30:05)
<code>%m</code>	The minutes part of time (30 for 11:30:05)
<code>%s</code>	The seconds part of time (05 for 11:30:05)
<code>%Y</code>	The year part of date (05 for Sept 6th 2005)
<code>%M</code>	The month part of date (09 for Sept 6th 2005)
<code>%D</code>	The day part of date (06 for Sept 6th 2005)
<code>%p</code>	The ncpus parameter
<code>%l</code>	The local_ncpus parameter
<code>%v</code>	The slave_ncpus parameter

Thus, if the parameter `PARALOOOP_lock_directory` is set to `lock-%Y-%M-%D`, the value of the lock directory selected at runtime will be: `lock-06-11-30` (for November 30th, 2006).

Input file specification:

With the switch `--input`. You may specify:

- an absolute path (/path/to/directory/input/seq.fasta)
- a relative path, taken from the current directory (input/seq.fasta)
- a file name: the file lives in the current directory (seq.fasta)

The special characters explained above may be included in the directories of file name.

Output and log files specification:

With the switch `--output`. Here, you only specify a prefix: each job will complete this prefix, generating a complete output file name and a complete log file name: for `ncpus=10`, 10 pairs of (output,log files) will be created in the output directory, with the following names:

- `BlastResult.0.0.txt,BlastResult.100.0.txt,BlastResult.200.0.txt,...`
- `BlastResult.0.0.log,BlastResult.100.0.log,BlastResult.200.0.log,...`

The first number is the first record number treated by this processor, the second number is incremented only if the `.txt` file grows too much (see parameter `PARALOOOP_max_file_size`).

Please note the prefix may include absolute or relative path specifications, as well as special characters, as explained above.

Load balancing:

When you launch 10 jobs or so in parallel, you have no warranty that some job does not last much more time than the others, in which case you loose a lot of time, waiting for only one processor to finish.

It is possible asking for some load balancing: in load balancing mode, when a job has finished his work, it tries to "steal" some work to the other jobs. If any other job has more than 1 record to work on (the number of records may be configured by the parameter `PARALOOOP_load_balancing_threshold`), the faster job steals half the remaining records to the slowest one. Load balancing is enabled by the parameter `PARALOOOP_load_balancing_enabled` or by the switch `load_balancing_enabled`

The log_level parameter

some information is logged to the log files. There are time stamps, returned values for the atomic jobs, etc. The log level may be adjusted with this parameter:

LOG LEVELS 0,01,012:

The value 0 logs nearly nothing. The default log level is 01, which logs more things. The complete logging is obtained by the 012 log level.

LOG LEVEL R:

Specifying an `R` in the log level parameters (together with the 0,1,2 characters, like in `012R`) leads to a special and very interesting behaviour: when the atomic job returns a value different from 0, meaning there is some problem, the input data is written to a file. The name of this file is the output file name, with the extension `.in` appended to it. This way, you have at the end of the processing the data which produced bad results: it is then very easy launching another paralooop job, probably with slightly different parameters, usqing those data as input file.

displaying the parameters:

It is sometimes difficult to know the value taken by every parameter, as there are several sources for setting their values. For this purpose, you can use the switch `--show_parameters`, together with the other switches and parameters: when `--show_parameters` is specified, paralooop starts as usual, but *instead executing any processing, it displays each values's parameter and stops*.

Specifying a paralooop run

All the following commands may be applied on an already run job, so you must specify the name of the job. This can be done in a few ways:

DEFAULT RUN:

If you don't specify anything, the job whose lock directory is has the default name `PARALOOP_lock` will be used.

LOCK DIRECTORY:

You may specify any lock directory, whatever its name is (the name of the lock directory may be chosen with the parameter `PARALOOP_lock_directory`)

JOB ID:

The job Id is a number from 1 to 10, if 10 cpus were requested. Say the lock directory is called `lock`, you may specify checking for the job nb 5 with: `paraloop.pl --check lock/5`

Checking your jobs:

Use `paraloop --check` followed by the lock directory name to display a screen telling you some information about the advancement of your jobs.

Id	start	end	step	size	current	adv	job Id	status	time
1	0	9	1	10	0	0	18069.node1d	RUNNING	000:00:00
2	10	18	1	9	10	0	17132.node2d	RUNNING	000:00:00

Interrupting your jobs

Use `paraloop --interrupt` followed by the lock directory name to interrupt your jobs. The jobs will be stopped only at the end of the current record, so that it will be easy to restart it.

Resuming your jobs

Use `paraloop --resume` followed by the lock directory name to resume your jobs after an interruption.

Restarting your jobs

Use `paraloop --restart` followed by the lock directory name to restart your jobs, after an interruption

RESTARTING VERSUS RESUMING:

Resuming the jobs means that the interrupted jobs are resumed from the point where they were interrupted. *Restarting* the jobs means that the jobs are restarted from the beginning: thus, the already computed data are computed again. However, no result is lost, because the already existing `log` and `output` files are renamed to `.bck`. Please note that in release 1.1 and 1.0, the `--restart` switch did in fact a *resume*.

The wait switch

Adding `--wait` to the above command line makes `paraloop` to wait for the end of all `paraloop` jobs. It is useful using this switch when `paraloop` is integrated to a script.

However, if you are tired of waiting for the end of the jobs (this can be long), you may type `ctrl-c` to retrieve your terminal. You can then restart `paraloop` in the wait mode, just typing:

```
paraloop.pl --wait lock
```

where `lock` is the lock directory. In this mode, `paraloop` does not make anything but waiting for the end of the jobs.

The clean switches:

Using the `--clean` switch, `paraloop` waits for the end of the process, then "cleans" your files and directories at the end of processing: the 10 output files (for `ncpus=10`) are all concatenated to a single output file, the log files and lock directory are removed.

WARNING, this is a somewhat dangerous switch, as it will be impossible to know if anything went wrong after the cleaning process: the log files are removed !

The verbose and quiet switches:

Adding `--verbose` to the command line is a good idea to display more messages.

The local switch:

If you specify `--local` to the paraloop command, the scheduler System is used, instead of any other scheduler: this causes the jobs to be executed on the local machine (hopefully a multiprocessor one), instead of being distributed by a `qsub` or any other protocol.

If the parameter `PARALOOP_no_local_mode` is specified, the `--local` switch cannot be used. This parameter is generally set by the administrator.

The list of allowed switches may be displayed with the command:
`paraloop.pl --switches`

Working with plugins:

Paraloop comes with 3 useful plugins. 2 of them belong to the bioinformatics field, the 3rd one is a general purpose plugin. You may write new plugins, as necessary for your job.

The list of installed plugins may be displayed with the command:
`paraloop.pl --plugins`

The Blast plugin:

This plugin performs a blast (ncbi or wu version) on each record read from the input file. `PARALOOP_Blast_params` let you specify parameters for the blast program: if you are using the ncbi version, `-p blastp` is the default value of this parameter. You

However, please note you do not have to specify the `-i`, `-o`, `-d` (for ncbi) switches through this parameter, as the plugin will specify the input file, the output file and the database in the correct way, whatever version of Blast (ncbi or wu) you are using.

<code>PARALOOP_Blast_origin</code>	ncbi or wu (default to ncbi)
<code>PARALOOP_Blast_path</code>	The path to the binary program (default <code>blastall</code> or <code>blastp</code>)
<code>PARALOOP_Blast_params</code>	The parameters used with the blast binary.
<code>PARALOOP_Blast_chunk</code>	Compute the sequences copying chunk files together in the input file. There is more performance than reading the sequences one by one.

The Shell plugin:

The Shell plugin considers each line of the input file as a line of an executable shell. You may start the line with a path to a shell interpreter, or consider that each line is interpreted by the shell whose name is set by the parameter `PARALOOP_Shell_interpreter`.

Here is an example showing the first method:

```

/bin/tcsh : blabla
/bin/perl : if ($TOTO==1){blabla;};

```

and here is an example showing the second method:

```

get_Blastx.pl --seqfile AC151527.13 --db SPTR --out AC151527.13blastx.gff
get_Blastx.pl --seqfile AC153128.1 --db SPTR --out AC153128.1blastx.gff
get_Blastx.pl --seqfile AC153000.5 --db SPTR --out AC153000.5blastx.gff
get_Blastx.pl --seqfile AC149809.10 --db SPTR --out AC149809.10blastx.gff
get_Blastx.pl --seqfile AC149305.18 --db SPTR --out AC149305.18blastx.gff

```

<code>PARALOOP_Shell_interpreter</code>	The path to the shell interpreter (default <code>/bin/sh</code>)
---	---

The Bioperl plugin:

The Bioperl plugin extends BpInput, ie the input file must be in a format readable by bioperl. Then, for each record, an external script is called. The path to this script is set through the parameter PARALOOOP_Bioperl_path. You'll have to write the external script, of course, but this can be a very simple script: let's suppose you want to seg a fasta database. You could write the following script, called seg_database:

```
#!/bin/sh
seq $2 > $3
exit 0;
```

For each record, the plugin creates a temporary input file, then calls your script with the command:

```
seg_database 0 input output
```

0 is the default value of the parameter PARALOOOP_Bioperl_params. It is passed to our script but not used here. Another version of seg_database, more powerful, could use this parameter: let's suppose we want to be able to pass some seg options to the script. We could write to some configuration file:

```
PARALOOOP_Bioperl_params="20 -a -x"
```

and rewrite our script as follows:

```
#!/bin/sh
seq $2 $1 > $3
exit 0;
```

For each record of the input fasta file, seg_database will be called as:

```
seg_database "20 -a -x" input output
```

PARALOOOP_Bioperl_path	The path to the external script
PARALOOOP_Bioperl_params	The parameter passed to this script
PARALOOOP_Bioperl_input_format	The input format read by the external script (generally not useful, as bioperl is able to recognize the input format)

Working with schedulers

The scheduler is the object which stands between the main program and the operating system. This object is responsible for distributing the jobs on the different processors, using some protocol.

Paralooop comes with 3 schedulers. You may write new schedulers if your architecture is not covered by any of them.

The list of installed schedulers may be displayed with the command:
paralooop.pl -schedulers

Selecting a scheduler:

You may select a scheduler with the parameter PARALOOOP_Scheduler. This is generally the job of the administrator to select the scheduler for the users. However, if a user calls paralooop with the --local switch, then the System scheduler is used, whatever value PARALOOOP_Scheduler is set to.

The System scheduler:

This scheduler is used in several situations:

- The computer is a multiprocessor computer without any queuing system installed. Paralooop uses fork calls to submit itself again when necessary.

- The user did specify the `--local` switch, bypassing any queuing system, and disabling any jobs distribution on the cluster, if any.
- We run in MASTER/SLAVE mode (cf. later).

PARALOOOP_Scheduler	System
---------------------	--------

The PBS scheduler

This scheduler is used when `paralooop` is ran upon a PBS² queuing system (or any PBS-compatible queuing system). Every new job is submitted through the `qsub` utility. The parameters for this scheduler are described in the table under. The account parameter may be specified through the `--account` switch.

The time limit

When used through the PBS scheduler, `paralooop` tries to know at each iteration if the time allowed for this job is finished. If so, `paralooop` launches a `qsub` command, submitting a new job, then interrupts its processing: this way, we can avoid being killed by the system because our cpu time limit is reached. Besides, you may specify a parameter called `PARALOOOP_fair_time_limit`: at each iteration, the elapsed time is compared to this parameter. If the elapsed time exceeds the fair time limit, a new job is submitted and the processing is interrupted.

PARALOOOP_Scheduler	PBS	
PARALOOOP_qsub_params	-l xxxx	string to pass to <code>qsub</code> .
PARALOOOP_account	name	String to pass to the <code>-A</code> switch of <code>qsub</code>
PARALOOOP_fair_time_limit	3600	See text

The Rsystem scheduler

This scheduler is used when `paralooop` lives in a cluster, but there is no specialized program to distribute the jobs: in this case, we must rely on some standard mechanism to distribute the jobs on the nodes. Rsystem calls `rsh` or `ssh` to submit `paralooop` on a distant node. You can describe the cluster in a very complete way; it is even possible defining subclusters, which can be useful for heterogeneous clusters. Here is a template configuration:

PARALOOOP_Scheduler	Rsystem	
PARALOOOP_Rsystem_rsh	ssh	The remote command (<code>rsh</code> or <code>ssh</code> , defaults to <code>rsh</code>)
PARALOOOP_Rsystem_tmp	/scratch	The temporary directory on each node (defaults to <code>/tmp</code>)
PARALOOOP_Rsystem_nodes	node1, node2, node3, node4, orange, tomate	The list of nodes belonging to the cluster
PARALOOOP_Rsystem_cluster_32	node1, node2	The nodes belonging to the subcluster 32 (any name may be used instead of 32)
PARALOOOP_Rsystem_cluster_64	node3, node4	The nodes belonging to the subcluster 64
PARALOOOP_Rsystem_cluster_lab	orange, tomate	The nodes belonging to the subcluster lab (in fact, some computers in the lab)
PARALOOOP_Rsystem_Node_orange	1, 1	1 cpu, arbitrary speed 1
PARALOOOP_Rsystem_Node_tomate	1, 2	1 cpu, but more powerful than orange
PARALOOOP_Rsystem_Node_node1	2, 1	2 cpus, not too powerful

PARALOOOP_Scheduler	Rsystem	
PARALOOOP_Rsystem_Node_node2	2, 1	
PARALOOOP_Rsystem_Node_node3	2, 2	2 cpus, powerful
PARALOOOP_Rsystem_Node_node4	2, 2	

Please note it is important using `ssh` rather than `rsh` if you are working with workstations anywhere in the lab. However, `ssh` must be configured so that no password will be asked (public key identification). the fair time limit parameter is not defined for this scheduler, because there is no way to ask for a later startup, as with queueing-based schedulers like PBS.

The autonomous or MASTER/SLAVE modes:

When in MASTER/SLAVE mode (parameter `PARALOOOP_mode`), only a master job is submitted through the scheduler. This master job then submits several slave jobs through another scheduler. The only tested configuration is: scheduler `PBS`, slave scheduler `system`. This is used with an SMP computer, using PBS but limiting the number of jobs a user may simultaneous run in the queue. The master job is launched through PBS, then many slave jobs are launched by this job. Here is a configuration template:

PARALOOOP_mode	MASTER/SLAVE	
PARALOOOP_Scheduler	PBS	tested only with this scheduler
PARALOOOP_slave_Scheduler	System	this is the scheduler used by the master to make the slaves to work. this is tested only with System
PARALOOOP_ncpus	1	The number of masters we launch in parallel. This is the number of PBS jobs simultaneously running
PARALOOOP_slave_ncpus	8	The number of slaves each master runs and monitors.
PARALOOOP_qsub_params	-l cput=48:00:00,ncpus=8	It may be necessary to specify the parameters of <code>qsub</code> , in order to be sure that the correct number of processors is allocated to the job.
PARALOOOP_fair_time_limit	18000	After 5 hours (about 40 hours of cpu time), the PBS job is finished and submitted again. This avoids being killed by PBS after 48 hours cpu time

Writing a new plugin

The Dummy plugin

To use `paraloop` for your calculations, you may use the `Shell` plugin and build an input file containing the commands you want to execute, or the `Bioperl` plugin and write a little script, called by this plugin for every input file record. However, in many situations, it is more convenient writing a new plugin to encapsulate the needed code. This is not a difficult task, and this chapter explains how to write such a plugin.

The first step is to decide if your plugin should extend the already existing `PdInput` or `LnInput` objects, or if you have to write also new Input-output routines:

- `BpInput` calls `bioperl` to read the input file. Thus, if the format of your files is a handled by `bioperl`, there is no problem using `BpInput`.
- `LnInput` considers that each line of the input file is a record: if your file is line-oriented, it is a good idea extending this object.

Writing an input object

Let's suppose it is impossible for you to use `BpInput` or `LnInput`. You'll have to write a new input object, let us call it `MyInput`. The best thing to do is to start from an existing object, let's say `LnInput`. So, please copy `LnInput.pm` to `MyInput.pm`

Module name, inheritance:

Please change `LnInput` to `MyInput` in the following lines, at start of the code:

```
package PARALOOP::PLUGIN::LnInput;

...
use Logger;
use PARALOOP::PLUGIN::Plugin;

@PARALOOP::PLUGIN::LnInput::ISA = qw( PARALOOP::PLUGIN::Plugin );
```

The `__Init` sub:

This sub is called by the `New` function (defined in the `Plugin.pm` object). Its main goal is to open the input resource and to initialize the private variable `__records_counter`.

The `NextRecord`, `Tell`, `TellLength` subs

`NextRecord` reads and returns the next record of the input file. It also updates `__records_counter`.

`Tell` returns `__records_counter`, and `TellLength` returns the number of records in the input file.

The `SkipRecords` sub

This sub skips the number of records passed by parameter. This number may be negative (this may be interesting when the user supplies a negative `--step` switch), even if for several plugins (namely `LnInput` and `BpInput`) this is forbidden. Do not forget to update `__records_counter`.

Writing the plugin

You can now write your plugin, extending the new object `MyInput`. Let's call it `MyPlugin`, first copy `Dummy.pm` to `MyPlugin.pm`

The `WhatParaloopPlugin` and `Parameter subs`

It is requisite to define `WhatParaloopPlugin`, because if this sub does not exist, the object you are writing is *not* considered as a plugin. `WhatParaloopPlugin` should return some lines shortly describing your plugin. The sub `parameters` should return the parameters needed by your plugin, their meanings and their default values in a string: it will be printed when the user will call `paraloop` with the `--parameters` switch.

The `__Init` and `DESTROY` subs

`__Init` is called by the constructor of the object. This function is important for liability of the plugin: its role is to check as much as possible, in order to be sure that the plugin will work when it will be given the processor (may be several hours after `paraloop` has been launched, if you are working with a queuing system). The general algorithm is described here:

- Retrieve the parameters: a ref to a `ParamParser` object, and a ref to a `Logger` object.
- Check we can open the input and output files
- Check the other parameters: do they have reasonable values ?
- Call the line:

```
$self->SUPER::__Init(-inputfile=>$inputfile,-log=>$log);
```

to initialize the superclass (generally a `XxInput` object, as described above).
- Ask the superclass the `job_id`, will be useful for temporary objects
- Create a temporary directory if necessary
- Store useful data in the private storage space of the object (`$self`)

The function `DESTROY` is called by perl when the program is terminated. It should remove every temporary directory or file crated by `__Init`.

The `Exec sub`

The computation you want to implement in your plugin is written in this sub. Its algorithm is explained here:

- `Exec` is passed only one parameter: the timeout in seconds, i.e. The max number of seconds the function `Exec` can spend.
- Retrieve useful variables from `%self`
- change to the temporary directory (created by `__Init`) if necessary.
- Read the next record of the input file, calling the `NextRecord` function of the `XxInput` object we extend. Remember this object is responsible for the input resource, this resource was opened by its `__Init` function.
- It may be useful to copy the record to a temporary file. As your current directory is now a temporary directory, created by `__Init` with an unique name, you do not have to care of the file name.
- Build a command line to call the external program with the correct parameters (`$cmd`). The output should be in a temporary file, as it is easier to compute the number of bytes returned by this call (stored in `$nb`).
- Call `$cmd` using the function `RunExt` from the module `Runner`:

```
my ($sts,$killed)=RunExt(-cmd=>$cmd,-timeout=>$timeout);
```
- The output file is not currently opened, however its name is available as a private variable: you can just copy the output of your processing to this file.
- Return 3 values: `$nb`, `$sts` (the returned value of your program), `$killed` (a number telling if a signal was received during the execution).

Debugging your plugin

You have a few solutions for debugging your plugin.

The `--plugin_debug` switch

To debug your plugin, you should start paralooop in monoprocessor mode, thus specifying the `--monoprocessor` switch, *but* specifying also the switch `--plugin_debug`. You also must specify a `--start` and `--end` switch. It is a good idea to specify a very short range of records .

Paralooop starts in *mono/init* mode, then instead of submitting a new `paralooop` through the scheduler, it switches to *mono/running* mode, calling the main loop. You may launch paralooop through the perl debugger (`perl -d paralooop.pl ...`), which makes easy to debug your code in the usual way.

The `--verbose` switch

You may also run paralooop with the `--verbose` switch, thus generating more informative messages.

The `log_level` parameter

In order to debug your plugin, do not forget logging informative messages through the logger object. You have to call the `$o_log->Trace` function to log something. You must specify a log level to this function: the normal level is 1, but it is good practice to log as many information as possible, may be with a log level of 2. Running `paralooop` with parameter `PARALOOOP_log_level` set to '012' will then produce a lot of messages , useful for debugging.