Paraloop 1.0: The documentation

Introduction

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

Copyright

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 to install 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.
2. 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

1 www.bioperl.org

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 manu prodom  4096 Apr 11 18:28 paraloop-1.0

cd paraloop-1.0; ls -l
total 12
drwxrwsr-x  2 manu prodom 4096 Apr 11 18:28 bin
drwxrwsr-x  2 manu prodom 4096 Apr 11 18:28 etc
drwxrwsr-x  3 manu prodom 4096 Apr 11 18:28 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 /home/user/paraloop/paraloop-1.0
```

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:

i. Using the whole path:
   `~/paraloop-1.0/bin/paraloop.pl`

ii. Defining an alias:
   `alias paraloop ~/paraloop-1.0/bin/paraloop.pl`

iii. Creating a link from a directory already in the path. For an admin:
   ```
   cd /usr/local/bin;
   ln -s  usr/local/paraloop-1.0/bin/paraloop.pl .
   ```

iv. 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:

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:

- `paraloop-1.0/etc/paraloop.root.cfg`
- `paraloop-1.0/etc/paraloop.cfg`

This is particularly important if you are installing paraloop 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 paraloop.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 paraloop.cfg, it is rather considered as a default value. The users may override it, if needed.

The paraloop user configuration:

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

- ~/.paralooprc This file will be always read by paraloop, 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 PARALOOP_error): in this case, you should write the following line in ~/.paralooprc:
  PARALOOP_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. For instance the blast parameters, used by the Blast plugin:
  PARALOOP_Blast_params = -p blastp -M BLOSUM62 -m 8

The list of configurable parameters may be displayed with the command:
paraloop.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 paraloop was called with the switch
-cfg f1.cfg,f2.cfg

1. The switch on the command line
2. .../etc/paraloop.root.cfg
3. f1.cfg
4. f2.cfg
5. $HOME/.paralooprc
6. .../etc/paraloop.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 root.cfg file cannot be reset afterwards, etc. The last file defines default values.

Using paraloop

How does paraloop work ?

Paraloop works in 5 different modes. Each mode is selected through the switches and the environment variable `$SUBCOUNT`.

1. *Mode "multi/init":* **paraloop** runs in "*multi/init*" mode if the switch -- `monoprocessor` is *not* specified. In this mode, paraloop just forks himself as many times as indicated through the `ncpus` parameter, selecting a convenient switches set for every monoprocessor instance.
2. *Mode "mono/init":* if the parameter `monoprocessor` is set and if the environment variable `$SUBCOUNT` is not set, `paraloop` runs in mode "`mono/init`": the parameters are read, then written back to a lock file, the needed plugin and scheduler are loaded and initialized in order to check the parameters, then `paraloop` sets the environment variable `$SUBCOUNT` and calls another instance of himself, through some submitting mechanism:
   - Calling `qsub` if a queueing system is installed (`PBS` scheduler).
   - Remote command (`rsh` ou `ssh`) if we live on a cluster without any queing system (`Rsystem` scheduler)
   - `fork` if we live on an SMP multiprocessor computer, or if the switch `--local` was specified (`System` scheduler).
3. *Mode "mono/running":* `paraloop` runs in "*mono/running*" mode when called, through a `qsub`, a `fork`, or a `ssh/rsh` utility, by another `paraloop` instance running in mode "*mono/init*" or "*mono/restart*". This is detected because the environment variable `$SUBCOUNT` is set. Its value indicates how many times `paraloop` submitted itself, because of a time parameter. In this mode, the main program loops, thus the wanted computations are executed.
4. Mode "*multi/restart*": `paraloop` runs in "*multi/restart*" mode when we restart all the interrupted `paraloop` instances, specifying only a directory through the `-restart` switch. `Paraloop` forks himself as many times as necessary, i.e. one instance per lock file found in the directory. Those instances will run in "*mono/restart*" mode.
5. Mode "*mono/restart*": `paraloop` runs in "*mono/restart*" mode when we restart only one instance of `paraloop`, specifying a lock file through the `-restart` switch. As in "*mono/init*" mode, `paraloop` submits itself through the appropriate mechanism. The new instance will run in "*mono/running*" mode.
6. Mode "*waiting*": when the switches `-wait` or `-waitonly` are specified, `paraloop` enters in the *waiting* mode: it makes nothing, just waiting until the other `paraloop` instances are finished.

Using paraloop:

Suppose you must 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:

```
paraloop.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 `paraloop` you want to use the plugin called Blast.

The start, end, interleaved parameters:

If you want to perform the blast only from record nb. 1000 to record nb. 10000 (included), you can specify the `--start` and `--end` switches.

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.
- *Interleaved mode*: sequence 1 to cpu 1, sequence 2 to cpu 2, etc.

The interleaved mode is selected with the `--interleaved` switch. The slice mode is the default mode (no switch to specify).

The verbose switch:

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

The wait switches

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

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

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. 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 vaulue 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.

`PARALOOP_Blast_origin`	`ncbi` or `wu` (default to ncbi)
`PARALOOP_Blast_path`	The path to the binary program (default `blastall` or `blastp`)
`PARALOOP_Blast_params`	The parameters used with the blast binary.
`PARALOOP_Blast_chunk`	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 on 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
```

`PARALOOP_Shell_interpreter`	The path to the shell interpreter (default `/bin/sh`)

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 `PARALOOP_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 PARALOOP_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:

```
PARALOOP_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, seq_database will be called as:

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

PARALOOP_Bioperl_path	The path to the external script
PARALOOP_Bioperl_params	The parameter passed to this script
PARALOOP_Bioperl_input_format	The input format for the external script

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.

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

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

Selecting a scheduler:

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

The System scheduler:

This scheduler is used in two situations:

- The computer is a multiprocessor computer without any queing system installed. Paraloop uses fork calls to submit itself again when necessary.
- The user did specify the --local switch, bypassing any queing system, and disabling any jobs distribution on the cluster, if any.

PARALOOP_Scheduler	Set to the value System

The PBS scheduler

This scheduler is used when paraloop is built upon a queing system like PBS[2] 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.

PARALOOP_Scheduler	Set to the value PBS
PARALOOP_qsub_params	string to pass to qsub. Ex: -a 1200 for starting the job at 12:00h
PARALOOP_account	String to pass to the -A switch of qsub

The Rsystem scheduler

This scheduler is used when paraloop 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 paraloop on a distant node.

PARALOOP_Scheduler	Set to the value Rsystem
PPARALOOP_Rsystem_nodes	The list of nodes belonging to the cluster
PARALOOP_Rsystem_rsh	The remote command (defaults to rsh)
PARALOOP_Rsystem_tmp	The temporary directory on each node (defaults to /tmp)

2 www.openpbs.org

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 queing 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 retuned 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 paraloop in monoprocessor mode, thus specifying the --monoprocessor switch, *but* specifying 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 .

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

The –verbose switch

You may also run paraloop 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 paraloop with parameter PARALOOP_log_level set to '012' will then produce a lot of messages , useful for debugging.