RAMPART Documentation

Release 0.11.0

Daniel Mapleson, Nizar Drou, David Swarbreck

March 17, 2016

Contents

1 Introduction **3**
- 1.1 Comparison to other systems . . . 4

2 Dependencies **5**

3 Installation **7**
- 3.1 Quick start . . . 7
- 3.2 From tarball . . . 7
- 3.3 From source . . . 8

4 Environment configuration **11**
- 4.1 Conan - scheduler configuration . . . 11
- 4.2 External process configuration . . . 11
- 4.3 Logging . . . 12

5 Running RAMPART **13**
- 5.1 The genome to assemble . . . 13
- 5.2 Defining datasets . . . 14
- 5.3 The pipeline . . . 15
- 5.4 Potential runtime problems . . . 26

6 Multi-sample projects **27**
- 6.1 Jellyswarm . . . 27
- 6.2 RAMPART multi-sample mode . . . 28

7 Citing RAMPART **31**

8 License **33**

9 Contact **35**

10 Acknowledgements **37**

RAMPART is a *de novo* assembly pipeline that makes use of third party-tools and High Performance Computing resources. It can be used as a single interface to several popular assemblers, and can perform automated comparison and analysis of any generated assemblies.

RAMPART was created by The Genome Analysis Centre (TGAC) in Norwich, UK by:

- Daniel Mapleson
- Nizar Drou
- David Swarbreck

Useful links:

- Project Website: http://www.tgac.ac.uk/rampart/
- Online Documentation (this manual): http://rampart.readthedocs.org/en/latest/index.html
- Source code and distributable tarball: https://github.com/TGAC/RAMPART

CHAPTER 1

Introduction

RAMPART is a configurable pipeline for *de novo* assembly of DNA sequence data. RAMPART is not a *de novo* assembler. There are already many very good freely available assembly tools, however, few will produce a good assembly, suitable for annotation and downstream analysis, first time around. The reason for this is that genome assembly of non-model organisms are often complex and involve tuning of parameters and potentially, pre and post processing of the assembly. There are many combinations of tools that could be tried and no clear way of knowing *a priori*, which will work best.

RAMPART makes use of tried and tested tools for read pre-processing, assembly and assembly improvement, and allows the user to configure these tools and specify how they should be executed in a single configuration file. RAMPART also provides options for comparing and analysing sequence data and assemblies.

This functionality means that RAMPART can be used for at least 4 different purposes:

- Analysing sequencing data and understanding novel genomes.
- Comparing and testing different assemblers and related tools on known datasets.
- An automated pipeline for *de novo* assembly projects.
- Provides a single common interface for a number of different assembly tools.

The intention is that RAMPART gives the user the possibility of producing a decent assembly that is suitable for distribution and downstream analysis. Of course, in practice not every assembly project is so straight forward, the actual quality of assembly is always going to be a function of at least these sequencing variables:

- sequencing quality
- sequencing depth
- read length
- read insert size

... and the genome properties such as:

- genome size
- genome ploidy
- genome repetitiveness

RAMPART enables a bioinformatician to get a reasonable assembly, given the constraints just mentioned, with minimal effort. In many cases, particularly for organisms with haploid genomes or relatively simple (i.e. not too heterozygous and not too repeaty) diploid genomes, where appropriate sequenceing has been conducted, RAMPART can produce an assembly that suitable for annotation and downstream analysis.

RAMPART is designed with High Performance Computing (HPC) resources in mind. Currently, LSF and PBS schedulers are supported and RAMPART can execute jobs in parallel over many nodes if requested. Having said this

RAMPART can be told to run all parts of the pipeline in sequence on a regular server provided enough memory is available for the job in question.

This documentation is designed to help end users install, configure and run RAMPART.

1.1 Comparison to other systems

Roll-your-own Make files This method probably offers the most flexibility. It allows you to define exactly how you want your tools to run in whatever order you wish. However, you will need to define all the inputs and outputs to each tool. And in some cases write scripts to manage interoperability between some otherwise incompatible tools. RAMPART takes all this complication away from the user as all input and output between each tool is managed automatically. In addition, RAMPART offers more support for HPC environments, making it easier to parallelize steps in the pipeline. Managing this manually is difficult and time consuming.

Galaxy This is a platform for chaining together tools in such a way as to promote reproducible analyses. It also has support for HPC environments. However, it is a heavy weight solution, and is not trivial to install and configure locally. RAMPART itself is lightweight in comparison, and ignoring dependencies, much easier to install. In addition, galaxy is not designed with *de novo* genome assembly specifically in mind, whereas RAMPART is. RAMPART places more constraints in the workflow design process as well as more checks initially before the workflow is started. In addition, as mentioned above RAMPART will automatically manage interoperability between tools, which will likely save the user time debugging workflows and writing their own scripts to manage specific tool interaction issues.

A5-miseq and **BugBuilder** Both are domain specific pipeline for automating assembly of microbial organisms. They are designed specifically with microbial genomes in mind and keep their interfaces simple and easy to use. RAMPART, while more complex to use, is far more configurable as a result. RAMPART also allows users to tackle eukaryote assembly projects.

iMetAMOS This is a configurable pipeline for isolate genome assembly and annotation. One distinct advantage of iMetAMOS is that it offers the ability to annotate your genome. It also supports some assemblers that RAMPART currently does not. Both systems are highly configurable, allowing the user to create bespoke pipelines and compare and validate the results of multiple assemblers. However, in it's current form, iMetAMOS doesn't have as much provision for automating or managing assembly scaffolding or gap filling steps in the assembly workflow. In addition, we would argue that RAMPART is more configurable, easier to use and has more support for HPC environments.

CHAPTER 2

Dependencies

In order to do useful work RAMPART can call out to a number of third party tools during execution. The current list of dependencies is shown below. For full functionality, all these tools should be installed on your environment, however they are not mandatory so you only need to install those which you wish to use.

Assemblers (RAMPART is not an assembler itself so you should have at least one of these installed to do useful work):

- Abyss V1.5
- ALLPATHS-LG V50xxx
- Platanus V1.2
- SPAdes 3.1
- SOAPdenovo V2
- Velvet V1.2
- Discovar V51xxx

Dataset improvement tools:

- Sickle V1.2
- Quake V0.3
- Musket V1.0

Assembly improvement tools:

- Platanus V1.2 (for scaffolding and gap closing)
- SSPACE Basic V2.0
- SOAP de novo V2 (for scaffolding)
- SOAP GapCloser V1.12
- Reapr V1.0

Assembly Analysis Tools:

- Quast V2.3 - for contiguity analysis
- CEGMA V2.4 - for assembly completeness analysis
- KAT V1.0 - for kmer analysis

Miscellaneous Tools:

- TGAC Subsampler V1.0 - for reducing coverage of reads

- Jellyfish V1.1.10 - for kmer counting
- Kmer Genie V1.6 - for determining optimal kmer values for assembly

To save time finding all these tools on the internet RAMPART provides two options. The first and recommended approach is to download a compressed tarball of all supported versions of the tools, which is available on the github releases page: `https://github.com/TGAC/RAMPART/releases`. The second option is to download them all to a directory of your choice. The one exception to this is SSPACE, which requires you to fill out a form prior to download. RAMPART can help with this. After the core RAMPART pipeline is compiled, type: `rampart-download-deps <dir>`. The tool will place all downloaded packages in a sub-directory called "rampart_dependencies" off of the specified directory. Note that executing this command does not try to install the tools, as this can be a complex process and you may wish to run a custom installation in order to compile and configure the tools in a way that is optimal for your environment.

In case the specific tool versions requested are no longer available to download the project URLs are specified below. It's possible that alternative (preferably newer) versions of the software may still work if the interfaces have not changed significantly. If you find that a tool does not work in the RAMPART pipeline please contact daniel.mapleson@tgac.ac.uk, or raise a job ticket via the github issues page: https://github.com/TGAC/RAMPART/issues.

Project URLs:

- Abyss - http://www.bcgsc.ca/platform/bioinfo/software/abyss
- ALLPATHS-LG - http://www.broadinstitute.org/software/allpaths-lg/blog/?page_id=12
- Cegma - http://korflab.ucdavis.edu/datasets/cegma/
- Discovar - http://www.broadinstitute.org/software/discovar/blog/
- KAT - http://www.tgac.ac.uk/kat/
- Kmer Genie - http://kmergenie.bx.psu.edu/
- Jellyfish - http://www.cbcb.umd.edu/software/jellyfish/
- Musket - http://musket.sourceforge.net/homepage.htm#latest
- Quake - http://www.cbcb.umd.edu/software/quake/
- Quast - http://bioinf.spbau.ru/quast
- Platanus - http://http://platanus.bio.titech.ac.jp/platanus-assembler/
- Reapr - https://www.sanger.ac.uk/resources/software/reapr/#t_2
- Sickle - https://github.com/najoshi/sickle
- SoapDeNovo - http://soap.genomics.org.cn/soapdenovo.html
- SOAP_GapCloser - http://soap.genomics.org.cn/soapdenovo.html
- SPAdes - http://spades.bioinf.spbau.ru
- SSPACE_Basic - http://www.baseclear.com/landingpages/basetools-a-wide-range-of-bioinformatics-solutions/sspacev12/
- Subsampler - https://github.com/homonecloco/subsampler
- Velvet - https://www.ebi.ac.uk/~zerbino/velvet/

CHAPTER 3

Installation

Before installing RAMPART please ensure any dependencies listed above are installed. In addition, the following dependencies are required to install and run RAMPART:

- Java Runtime Environment (JRE) V1.7+

RAMPART can be installed either from a distributable tarball, from source via a `git clone`, or via homebrew. These steps are described below. Before that however, here are a few things to keep in mind during the installation process:

3.1 Quick start

To get a bare bones version of RAMPART up and running quickly, we recommend installation via Homebrew. This requires you to first install homebrew and tap homebrew/science. On Mac you can access homebrew from `http://brew.sh` and linuxbrew for linux from `https://github.com/Homebrew/linuxbrew`. Once installed make sure to tap homebrew science with `brew tap homebrew/science`. Then, as discussed above, please ensure you have JRE V1.7+ installed. Finally, to install RAMPART simply type `brew install rampart`. This will install RAMPART into your homebrew cellar, with the bare minimum of dependencies: Quake, Kmergenie, ABySS, Velvet, Quast, KAT.

3.2 From tarball

RAMPART is available as a distributable tarball. The installation process is simply involves unpacking the compressed tarball, available from the RAMPART github page: `https://github.com/TGAC/RAMPART/releases`, to a directory of your choice: `tar -xvf <name_of_tarball>`. This will create a directory called `rampart-<version>` and in there should be the following sub-directories:

- bin - contains the main rampart script and other utility scripts
- doc - a html and pdf copy of this manual
- etc - contains default environment configuration files, and some example job configuration files
- man - contains a copy of the manual in man form
- repo - contains the java classes used to program the pipeline
- support_jars - contains source and javadocs for the main rampart codebase

Should you want to run the tools without referring to their paths, you should ensure the 'bin' sub-directory is on your PATH environment variable.

Also please note that this method does not install any dependencies automatically. You must do this before trying to run RAMPART.

3.3 From source

RAMPART is a java 1.7 / maven project. Before compiling the source code, please make sure the following dependencies are installed:

- GIT
- Maven 3
- JDK v1.7+
- Sphinx and texlive (If you would like to compile this documentation. If these are not installed you must comment out

the `create-manual` execution element from the pom.xml file.)

You also need to make sure that the system to are compiling on has internet access, as it will try to automatically incorporate any required java dependencies via maven. Now type the following:

```
git clone https://github.com/TGAC/RAMPART.git
cd RAMPART
mvn clean install
```

Note: If you cannot clone the git repositories using "https", please try "ssh" instead. Consult github to obtain the specific URLs.

Assuming there were no compilation errors. The build, hopefully the same as that described in the previous section, can now be found in `./build/rampart-<version>`. There should also be a `dist` sub directory which will contain a tarball suitable for installing RAMPART on other systems.

Some common errors the user may encounter, and steps necessary to fix the, during the installation procedure follow:

1. Old Java Runtime Environment (JRE) installed:

```
Exception in thread "main" java.lang.UnsupportedClassVersionError:
uk/ac/tgac/rampart/RampartCLI : Unsupported major.minor version 51.0
```

This occurs When trying to run RAMPART, or any associated tools, with an old JRE version. If you see this, install or load JRE V1.7 or later and try again. Note that if you are trying to compile RAMPART you will need JDK (Java Development Kit) V1.7 or higher as well.

2. Incorrect sphinx configuration:

If you are compiling RAMPART from source, if you encounter a problem when creating the documention with sphinx it maybe that your sphinx version is different from what is expected. Specifically please check that your sphinx configuration provides a tool called `sphinx_build` and that tool is available on the PATH. Some sphinx configurations may have an executable named like this `sphinx_build_<version>`. If this is the case you can try making a copy of this executable and remove the version suffix from it. Alternatively if you do not wish to compile the documentation just remove it by commenting out the `create-manual` element from the pom.xml file.

3. Texlive not installed:

```
"[ERROR] Failed to execute goal org.apache.maven.plugins:maven-antrun-plugin:1.7:run
(create-manual) on project rampart:  An Ant BuildException has occured:
Warning:  Could not find file RAMPART.pdf to copy." Creating an empty
RAMPART.pdf file in the specified directory fixed this issue, allowing RAMPART
to successfully build
```

This error occurs because the RAMPART.pdf file was not created when trying to compile the documentation. RAMPART.pdf is created from the documentation via sphinx and texlive. If you see this error then probably sphinx is working fine but texlive is not installed. Properly installing and configuring texlive so it's available on your path should fix this issue. Alternatively if you do not wish to compile the documentation just remove it by commenting out the `create-manual` element from the pom.xml file.

CHAPTER 4

Environment configuration

RAMPART is designed to utilise a scheduled environment in order to exploit the large-scale parallelism high performance computing environments typically offer. Currently, LSF and PBS schedulers are supported, although it is also possible to run RAMPART on a regular server in an unscheduled fashion.

In order to use a scheduled environment for executing RAMPART child jobs, some details of your specific environment are required. These details will be requested when installing the software, however, they can be overwritten later. By default the settings are stored in `etc` folder within the project's installation/build directory, and these are the files that will be used by RAMPART by default. However, they can be overridden by either keeping a copy in `~/.tgac/rampart/` or by explicity specifying the location of the files when running RAMPART. Priority is as follows:

- custom configuration file specified at runtime via the command line - `--env_config=<path_to_env_config_file>`
- user config directory - `~/.tgac/rampart/conan.properties`
- installation directory - `<installation dir>/etc/conan.properties`

4.1 Conan - scheduler configuration

RAMPART's execution context is specified by default in a file called "conan.properties". In this file it is possible to describe the type of scheduling system to use and if so, what queue to run on. Valid properties:

- `executionContext.scheduler` = Valid options {"","LSF","PBS","SLURM"}
- `executionContext.scheduler.queue` = The queue/partition to execute child jobs on.
- `executionContext.locality = LOCAL` Always use this for now! In the future it may be possible to execute child jobs at a remote location.
- `externalProcessConfigFile = <location to external process loading file>` See next section for details of how to setup this file.

Lines starting with # are treated as comments.

4.2 External process configuration

RAMPART can utilise a number of dependencies, each of which may require modification of environment variables in order for it to run successfully. This can be problematic if multiple versions of the same piece of software need to be available on the same environment. At TGAC we execute python scripts for configure the environment for a tool,

although other institutes may use an alternative system like "modules". Instead of configuring all the tools in one go, RAMPART can execute commands specific to each dependency just prior to it's execution. Currently known process keys are described below. In general the versions indicated have been tested and will work with RAMPART, however other versions may work if their command line interface has not changed significantly from the listed versions. Note that these keys are hard coded, please keep the exact wording as below, even if you are using a different version of the software. The format for each entry is as follows: `<key>=<command_to_load_tool>`. Valid keys:

```
# Assemblers
Abyss_V1.5
AllpathsLg_V50
Platanus_Assemble_V1.2
SOAP_Assemble_V2.4
Spades_V3.1
Velvet_V1.2

# Dataset improving tools
Sickle_V1.2
Quake_V0.3
Musket_V1.0

# Assembly improving tools
Platanus_Gapclose_V1.2
Platanus_Scaffold_V1.2
SSPACE_Basic_v2.0
SOAP_GapCloser_V1.12
SOAP_Scaffold_V2.4
Reapr_V1
FastXRC_V0013

# Assembly analysis tools
Quast_V2.3
Cegma_V2.4
KAT_Comp_V1.0
KAT_GCP_V1.0
KAT_Plot_Density_V1.0
KAT_Plot_Spectra-CN_V1.0

# Misc tools
Jellyfish_Count_V1.1
Jellyfish_Merge_V1.1
Jellyfish_Stats_V1.1
Subsampler_V1.0
KmerGenie_V1.6
```

By default RAMPART assumes the tools are all available and properly configured. So if this applies to your environment then you do not need to setup this file.

4.3 Logging

In addition, RAMPART uses SLF4J as a logging facade and is currently configured to use LOG4J. If you which to alter to logging configuration then modify the "log4j.properties" file. For details please consult: "http://logging.apache.org/log4j/2.x/"

CHAPTER 5

Running RAMPART

Running RAMPART itself from the command line is relatively straight forward. The syntax is simply: `rampart [options] <path_to_job_configuration_file>`. To get a list and description of available options type: `rampart --help`. Upon starting RAMPART will search for a environment configuration file, a logging configuration file and a job configuration file. RAMPART will configure the execution environment as appropriate and then execute the steps specified in the job configuration file.

Setting up a suitable configuration file for your assembly project is more complex however, and we expect a suitable level of understanding and experience of *de novo* genome assembly, NGS and genome analysis. From a high level the definition of a job involves supplying information about 3 topics: the organism's genome; the input data; and how the pipeline should execute. In addition, we recommend the user specifies some metadata about this job for posterity and for reporting reasons.

The job configuration file must be specified in XML format. Creating a configuration file from scratch can be daunting, particularly if the user isn't familiar with XML or other markup languages, so to make this process easier for the user we provide a number of example configuration files which can be modified and extended as appropriate. These can be found in the `etc/example_job_configs` directory. Specifically the file named `ecoli_full_job.xml` provides a working example configuration file once you download the raw reads from: `http://www.ebi.ac.uk/ena/data/view/DRR015910`.

5.1 The genome to assemble

Different genomes have different properties. The more information provided the better the analysis of the assembly and the more features are enabled in the RAMPART pipeline. As a minimum the user must enter the name of the genome to assemble this is used for reporting and logging purposes and if a prefix for name standardisation isn't provided initials taken from the name are used in the filename and headers of the final assembly. In addition it is recommended that the user provides the genome ploidy as this is required if you choose to run the ALLPATHS-LG assembler. It is also useful for calculating kmer counting hash sizes, analysing the assemblies. If you set to "2", i.e. diploid, then assembly enhancement tools may make use of bubble files if they are available and if the tools are capable. If you plan to assemble polyploid genomes please check that the third-party tools also support this. ALLPATHs-LG for example cannot currently assembly polyploid genomes for example. An example XML snippet containing this basic information is shown below:

```
<organism name="Escherichia coli" ploidy="1"/>
```

If you plan to assemble a variant of an organism that already has a good quality assembly then the user should also provide the fasta file for that assembly. This allows RAMPART to make better assessments of assemblies produced as part of the pipeline and enables input read subsampling (coverage reduction). It also is used to make better estimates of required memory usage for various tools. An example XML snippet containing a genome reference is shown below:

```
<organism name="Escherichia coli" ploidy="1">
   <reference name="EcoliK12" path="EcoliK12.fasta"/>
</organism>
```

If you are assembling a non-model organism an suitable existing reference may not be available. In this case it is beneficial if you have any expectations of genomic properties that you provide them in order to make better assembly assessments and enable input read subsampling an memory requirement estimation. Estimated input values can be entered as follows. Note that these are optional, so the user can specify any or all of the properties as known, although the estimated genome size is particularly useful:

```
<organism name="Escherichia coli" ploidy="1">
   <estimated est_genome_size="4600000" est_gc_percentage="50.8" est_nb_genes="4300"/>
</organism>
```

5.2 Defining datasets

Before an assembly can be made some sequencing data is required. Sometimes an modern assembly project might involve a single set of sequencing data, othertimes it can involve a number of sequencing projects using different protocols and different data types. In order to instruct the assemblers and other tools to use the data in the right way, the user must describe each dataset and how to interpret it.

Each dataset description must contain the following information:

- Attribute “name” - an identifier - so we can point tools to use a specific dataset later.
- Element “files” - Must contain one of more “path” elements containing file paths to the actual sequencing data. For paired end and mate pair datasets this will involve pointers to two separate files.

Ideally you should specify the following information as well if you want RAMPART to execute all tools with the best settings:

- Attribute “read_length” -The length of each read in base pairs.
- Attribute “avg_insert_size” - An estimate of the average insert size in base pairs used for sequencing if this is a paired end or mate pair library.
- Attribute “insert_err_tolerance” - How tolerant tools should be when interpreting the average insert size specified above. This figure should be a percentage, e.g. the tool should accept inserts sizes with a 30% tolerance either side of the average insert size.
- Attribute “orientation” - If this is a paired end or mate pair library, the orientation of the reads. For example, paired end libraries are often created using “forward reverse” orientation, and often long mate pairs use “reverse forward” orientation. The user should specify either “FR” or “RF” for this property.
- Attribute “type” - The kind of library this is. Valid options: “SE” - single end; “OPE” - overlapping paired end; “PE” - paired end; “MP” - mate pair.
- Attribute “phred” - The ascii offset to apply to the quality scores in the library. Valid options: “PHRED_33” (Sanger / Illumina 1.8+); “PHRED_64” (Illumina 1.3 - 1.7).
- Attribute “uniform” - Whether or not the reads have uniform length. This is set to true by default. This property is used to work out the fastest way to calculate the number of bases present in the library for downsampling, should that be requested.

An example XML snippet of a set of NGS datasets for an assembly project are shown below:

```
<libraries>
    <library name="pe1" read_length="101" avg_insert_size="500" insert_err_tolerance="0.3"
             orientation="FR" type="PE" phred="PHRED_64">
```

```
        <files>
            <path>lib1_R1.fastq</path>
            <path>lib1_R2.fastq</path>
        </files>
    </library>
    <library name="mp1" read_length="150" avg_insert_size="4000" insert_err_tolerance="0.3"
             orientation="RF" type="MP" uniform="false" phred="PHRED_64">
        <files>
            <path>lib2_R1.fastq</path>
            <path>lib2_R2.fastq</path>
        </files>
    </library>
    <library name="ope1" read_length="101" avg_insert_size="180" insert_err_tolerance="0.3"
            orientation="FR" type="OPE" phred="PHRED_33">
        <files>
            <path>lib3_R1.fastq</path>
            <path>lib3_R2.fastq</path>
        </files>
    </library>
</libraries>
```

In the future we plan to interrogate the libraries to work out many of the settings automatically. However, for the time being we request that you enter all these details manually.

5.3 The pipeline

The RAMPART pipeline can be separated into a number of stages, all of which are optional customisable. The pipeline can be controlled in two ways. The first way is by definition in the job configuration file. If a pipeline stage is not defined it will not be executed. The second way is via a command line option: `-s`. By specifying which stages you wish to execute here you can run specific stage of the pipeline in isolation, or as a group. For example by typing: `rampart -s MECQ,MASS job.cfg`, you instruct RAMPART to run only the MECQ and MASS stages described in the job.cfg file. A word of caution here, requesting stages not defined in the configuration file does not work. Also you must ensure that each stage has it's pre-requisites fulfilled before starting. For example, you cannot run the AMP stage, without a selected assembly to work with.

5.3.1 MECQ - Multiple Error Correction and Quality Trimming Tool

The purpose of this step is to try to improve the input datasets. The user can select from a number of seperate tools can be executed on one or more of the input datasets provided. The user can also request whether or not the tools should be run linearly or in parallel.

Attempting to improve the dataset is a slightly controversial topic. Although, it is true that having good quality data is critical to creating a good assembly, then benefits from trimming and correcting input data are debatable. It is certainly true that error correction tools in particular can boost assembly contiguity, however this can occasionally come at the expense of missassemblies. In addition, trimming reads can alter the kmer-coverage statistics for the dataset and in turn confuse assemblers into making incorrect choices.

It is also worth noting that some assemblers perform their own error correction, for example, ALLPATHS-LG and SPAdes. Meaning that additional error correction via tools such as quake would not be significantly beneficial. It is a complicated topic and the decision to quality trim and error correct reads is left to the user. RAMPART makes it simpler to incorperate this kind of process into the assembly pipeline for assemblers which don't already have this option built in. The authors advice is to learn the error correction and assembly tools well, understand how they work and understand your data. Finally, RAMPART offers a suitable platform for testing these combinations out, so if you have the time and computational resources, it might be worth experimenting with different permutations.

Whilst new tools will be added as and when the are needed, currently MECQ supports the following tools:

- Sickle V1.1
- Quake V0.3
- Musket V1.0
- TrimGalore V

An example XML snippet demonstrating how to run two different tools in parallel, one on two datasets, the other on a single dataset:

```
<mecq parallel="false">
   <ecq name="sickle_agressive" tool="SICKLE_V1.2" libs="lib1896-pe,lib1897-mp"/>
   <ecq name="quake" tool="QUAKE_V0.3" libs="lib1896-pe1"
        threads="4" memory="8000"/>
</mecq>
```

MECQ produces output in the `mecq` directory produced in the specified job output directory. The directory will contain sub-directories relating to each `ecq` element described in the XML snippet, then further sub-directories relating to the specified libraries used for that `ecq`. The next steps in the pipeline (KMER and MASS) know how to read this directory structure to get their input automatically.

Adding other command line arguments to the error corrector

MECQ offers two ways to add command line arguments to the error corrector. The first is via a POSIX format string containing command line options/arguments that should be checked/validated as soon as the configuration file is parsed. Checked arguments undergo a limited amount of validation to check the argument name is recognized and that the argument values (if required) are plausible. The second method is to add a string containing unchecked arguments directly to the assembler verbatim. This second method is not recommended in general because any syntax error in the options will only register once the assembler starts running, which maybe well into the workflow. However, it is useful for working around problems that can't be easily fixed in any other way. For example, checked args only work if the developer has properly implemented handling of the argument in the error corrector wrapper. If this has not been implemented then the only way to work around the problem is to use unchecked arguments.

The following example demonstrates how to set some checked and unchecked arguments for Quake:

```
<mecq>
   <ecq name="quake" tool="QUAKE_V0.3" threads="16" memory="16000"
         libs="lib1896-pe1"
         checked_args="-k 19 -q 30 --hash_size=10000000"
         unchecked_args="-l --log"/>
   </job>
</mass>
```

Note that we use POSIX format for the checked arguments, regardless of what the underlying tool typically would expect. Unchecked arguments are passed verbatim to the tool.

You should also ensure that care is taken not to override variables, otherwise unpredictable behaviour will occur. In general options related to input libraries, threads/cpus and memory values are set separately. Also take care not to override the same options in both checked_args and unchecked_args.

5.3.2 Analysing reads

This stage analyses all datasets, both the RAW and those, if any, which have been produced by the MECQ stage.

Currently, the only analysis option provided involves a kmer analysis, using tools called jellyfish and KAT. This process will produce GC vs kmer frequence plots, which can highlight potential contamination and indicate whether you have sufficient coverage in your datasets for assembling your genome.

The user has the option to control, the number of threads and amount of memory to request per process and whether or not the kmer counting for each dataset should take place in parallel. An example of this is shown below:

```
<analyse_reads kmer="true" parallel="true" threads="16" memory="4000"/>
```

Note: This step is required if you wish to count kmers in the assemblies and compare the kmer content of reads to assemblies. See _ref::mass for more details.

5.3.3 MASS - Multiple Assembly Creation

This tool enables the user to try different assemblers with different settings. Currently, the following assemblers are supported by RAMPART (brackets below indicate tool name to use if config file - case insensitive):

- Abyss V1.5 (ABYSS_V1.5)
- ALLPATHS-LG V50xxx (ALLPATHSLG_V50)
- Platanus V1.2 (Platanus_Assemble_V1.2)
- SOAP denovo V2.04 (SOAP_Assemble_V2.4)
- SPAdes V3.1 (Spades_V3.1)
- Velvet V1.2 (Velvet_V1.2)
- Discovar V51xxx (Discovar_V51XXX)

A simple MASS job might be configured as follows:

```
<kmer_calc threads="32" memory="20000"/>
<mass>
   <job name="abyss-raw-kmer" tool="ABYSS_V1.5" threads="16" memory="4000" exp_walltime="60">
      <inputs>
         <input ecq="raw" lib="pe1"/>
      </inputs>
   </job>
</mass>
```

This instructs RAMPART to run a single Abyss assembly using 16 threads, requesting 4GB RAM, expecting to run for 60mins, using the optimal kmer value determined by kmer genie on the raw pe1 dataset. The kmer_calc stage looks ahead to run on dataset configurations for each MASS job.

In the job element there are two required attributes: “name” and “tool”. The name attribute is primarily used as the name of the output directory for this job, but it also provides a way of referring to this job from other parts of the pipeline. The tool attribute must represent one of the supported assemblers, and take one of the assemblers values defined at the start of this chapter, or in the environment config section of the documentation.

There are also several optional attributes: “threads”, “memory”, “exp_walltime”, “checked_args”, “unchecked_args”. The value entered to threads will be passed to the tool and the scheduler to define the number of threads required for this job. memory may get passed to the tool, depending on whether the tool requires it, but will get passed to the scheduler. exp_walltime, will just go to the scheduler. It’s important to understand how your scheduler works before entering these values. The intention is that these figures will represent guidelines to help the scheduler organise it’s workload fairly, such as LSF. However other schedulers may define these as hardlimits. For example on PBS there is no notion of “expected” walltime, only a hard limited walltime, so we double the value entered here in order to create a conservative hard limit instead. checked and unchecked args are described later in this section.

Varying kmers for De Bruijn Graph assemblers

Many DeBruijn graph assemblers require you to specify a parameter that defines the kmer size to use in the graph. It is not obvious before running the assembly which kmer value will work best and so a common approach to the problem is to try many kmers to "optimise" to the kmer parameter. RAMPART allows the user to do this in two different ways.

First, RAMPART supports kmergenie. If the user enters the kmergenie element in the mass element then kmer genie is used to determine the best kmer values to use for each distinct mass configuration. For example, if the same single dataset is used for each mass job then kmergeneie is run once, and that optimal kmer value is passed on to each mass job. If different datasets are used for building contigs in different mass jobs then RAMPART will automatically work out in which combinations of kmer genie need to be run to drive the pipeline.

The alternative way is to manually specify which kmer to use or to request a kmer spread, i.e. to define the range of kmer values that should be tried. This maybe necessary, if for example you would like to do your own analysis of the resultant assemblies, or if kmer genie fails on your dataset. If the user specifies both kmergenie and a manual kmer spread, then the manual kmer spread will override the kmergenie recommendation. The snippet below shows how to run Abyss using a spread of kmer values:

```
<mass>
   <job name="abyss-raw-kmer" tool="ABYSS_V1.5" threads="16" memory="4000">
      <kmer min="61" max="101" step="COARSE"/>
      <inputs>
         <input ecq="raw" lib="pe1"/>
      </inputs>
   </job>
</mass>
```

As you can see the XML element starting `<kmer` has been modified to specify a min, max and step value. Min and max obviously set the limits of the kmer range. You can omit the min and/or max values. If so the default min value is set to 35 and the default max value will be automatically determined by the provided input libraries. Specifically, the default max K will be 1 less than the read length of the library with smallest read length.

The step value, controls how large the step should be between each assembly. The valid options include any integer between 2 and 100. We also provide some special keywords to define step size: `FINE,MEDIUM,COARSE`, which correspond to steps of `4,10,20` respectively. Alternatively, you can simply specify a list of kmer values to test. The following examples all represent the same kmer range (61,71,81,91,101):

```
<kmer min="61" max="101" step="10"/>
<kmer min="61" max="101" step="MEDIUM"/>
<kmer list="61,71,81,91,101"/>
```

Note: Depending on the assembler used the values specified for the kmer range, the actual assemblies generated may be executed with slightly different values. For example some assemblers do not allow you to use kmers of even value. Others may try to optimise the k parameter themselves. We therefore make a best effort to match the requested RAMPART kmer range to the actual kmer range executed by the assembler.

Assemblers such as SPAdes and Platanus have their own K optimisation strategies. In these cases, instead of running multiple instances of these assemblers, RAMPART will run a single instance, and translate the kmer range information into the parameters suitable for these assemblers.

Some De Bruijn graph assemblers, such as ALLPATHS-LG, recommend that you do not modify the kmer value. In these cases RAMPART lets the assembler manage the k value. If the selected assembler does require you to specify a k value, and you omit the kmer element from the config, then RAMPART specifies a default kmer spread for you. This will have a min value of 35, the max is automatically determined from the provided libraries as described above, and the step is 20 (COARSE).

Varying coverage

In the past, sequencing was expensive and slow, which led to sequencing coverage of a genome to be relatively low. In those days, you typically would use all the data you could get in your assembly. These days, sequencing is relatively cheap and it is often possible to over sequence data, to the point where the gains in terms of separating signal from noise become irrelevant. Typically, a sequencing depth of 100X is more than sufficient for most purposes. Furthermore, over sequencing doesn't just present problems in terms of data storage, RAM usage and runtime, it also can degrade the quality of some assemblies. One common reason for failed assemblies with high coverage can occur if trying to assemble DNA sequenced from populations rather than a single individual. The natural variation in the data can make it impossible to construct unambiguous seed sequences to start a *De Brujin* graph.

Therefore RAMPART offers the ability to randomly subsample the reads to a desired level of coverage. It does this either by using the assembler's own subsampling functionality if present (ALLPATHS-LG does have this functionality), or it will use an external tool developed by TGAC to do this if the assembler doesn't have this functionality. In both cases user's interface to this is identical, and an example is shown below:

```
<mass>
   <job name="abyss-raw-cvg" tool="ABYSS_V1.5" threads="16" memory="4000">
      <coverage min="50" max="100" step="MEDIUM" all="true"/>
      <inputs>
         <input ecq="raw" lib="pe1"/>
      </inputs>
   </job>
</mass>
```

This snippet says to run Abyss varying the coverage between 50X to 100X using a medium step. It also says to run an abyss assembly using all the reads. The step options has the following valid values: `FINE, MEDIUM, COARSE`, which correspond to steps of: `10X, 25X, 50X`. If the user does not wish to run an assembly with all the reads, then they should set the all option to false.

Varying other variables

MASS provides a mechanism to vary most parameters of any assembler. This is done with the `var` element, and there can be only one `var` element per MASS job. The parameter name should be specified by an attribute called `name` in that element and the values to test should be put in a single comma separated string under an attribute called `values`. For example, should you wish to alter the coverage cutoff parameter in the velvet assembler you might write something like this:

```
<mass>
   <job name="velvet-cc" tool="VELVET_V1.2" threads="16" memory="8000">
      <kmer list="75"/>
      <var name="cov_cutoff" values="2,5,10,auto"/>
      <inputs>
         <input ecq="raw" lib="pe1"/>
      </inputs>
   </job>
</mass>
```

Note that in this example we set the kmer value to 75 for all tests. If the kmer value is not specified then the default for the assembler should be used.

Using multiple input libraries

You can add more than one input library for most assemblers. You can specify additional libraries to the MASS job by simply adding additional `input` elements inside the `inputs` element.

MASS supports the ALLPATHS-LG assembler, which has particular requirements for its input: a so-called fragment library and a jumping library. In RAMPART nomenclature, we would refer to a fragment library, as either an overlapping paired end library, and a jumping library as either a paired end or mate pair library. ALLPATHS-LG also has the concept of a long jump library and long library. RAMPART will translate mate pair libraries with an insert size > 20KBP as long jump libraries and single end reads longer than 500BP as long libraries.

An simple example of ALLPATHS-LG run, using a single fragment and jumping library is shown below:

```
<mass>
   <job name="allpaths-raw" tool="ALLPATHSLG_V50" threads="16" memory="16000">
      <inputs>
         <input ecq="raw" lib="pe1"/>
         <input ecq="raw" lib="mp1"/>
      </inputs>
   </job>
</mass>
```

Multiple MASS runs

It is possible to ask MASS to conduct several MASS runs. You may wish to do this for several reasons. The first might be to compare different assemblers, another reason might be to vary the input data being provided to a single assembler.

The example below shows how to run a spread of Abyss assemblies and a single ALLPATHS assembly on the same data:

```
<mass parallel="true">
   <job name="abyss-raw-kmer" tool="ABYSS_V1.5" threads="16" memory="4000">
      <kmer min="65" max="85" step="MEDIUM"/>
      <inputs>
         <input ecq="raw" lib="pe1"/>
         <input ecq="raw" lib="mp1"/>
      </inputs>
   </job>
   <job name="allpaths-raw" tool="ALLPATHSLG_V50" threads="16" memory="16000">
      <inputs>
         <input ecq="raw" lib="pe1"/>
         <input ecq="raw" lib="mp1"/>
      </inputs>
   </job>
</mass>
```

Note that the attribute in MASS called `parallel` has been added and set to true. This says to run the Abyss and ALLPATHS assemblies in parallel in your environment. Typically, you would be running on a cluster or some other HPC architecture when doing this.

The next example, shows running two sets of abyss assemblies (not in parallel this time) each varying kmer values in the same way, but one set running on error corrected data, the other on raw data:

```
<mass parallel="false">
   <job name="abyss-raw-kmer" tool="ABYSS_V1.5" threads="16" memory="4000">
      <kmer min="65" max="85" step="MEDIUM"/>
      <inputs>
         <input ecq="raw" lib="pe1"/>
      </inputs>
   </job>
   <job name="abyss-raw-kmer" tool="ABYSS_V1.5" threads="16" memory="4000">
      <inputs>
```

```
         <input ecq="quake" lib="pe1"/>
      </inputs>
   </job>
</mass>
```

Adding other command line arguments to the assembler

MASS offers two ways to add command line arguments to the assembler. The first is via a POSIX format string containing command line options/arguments that should be checked/validated as soon as the configuration file is parsed. Checked arguments undergo a limited amount of validation to check the argument name is recognized and that the argument values (if required) are plausible. The second method is to add a string containing unchecked arguments directly to the assembler verbatim. This second method is not recommended in general because any syntax error in the options will only register once the assembler starts running, which maybe well into the workflow. However, it is useful for working around problems that can't be easily fixed in any other way. For example, checked args only work if the developer has properly implemented handling of the argument in the assembler wrapper script. If this has not been implemented then the only way to work around the problem is to use unchecked arguments.

The following example demonstrates how to set some checked and unchecked arguments for Abyss:

```
<mass>
   <job name="abyss" tool="ABYSS_V1.5" threads="16" memory="16000"
         checked_args="-n 20 -t 250"
         unchecked_args="p=0.8 q=5 s=300 S=350">
      <kmer list="83"/>
      <inputs>
         <input ecq="raw" lib="ope1"/>
         <input ecq="raw" lib="mp1"/>
      </inputs>
   </job>
</mass>
```

Note that we use POSIX format for the checked arguments, regardless of what the underlying tool typically would expect. Unchecked arguments are passed verbatim to the tool.

You should also ensure that care is taken not to override variables, otherwise unpredictable behaviour will occur. In general options related to input libraries, threads/cpus, memory and kmer values are set separately. Also remember not to override arguments that you may be varying using a `var` element.

Navigating the directory structure

Once MASS starts it will create a directory within the job's output directory called `mass`. Inside this directory you might expect to see something like this:

```
- <Job output directory>
-- mass
--- <mass_job_name>
---- <assembly> (contains output from the assembler for this assembly)
---- ...
---- unitigs (contains links to unitigs for each assembly and analysis of unitigs)
---- contigs (contains links to contigs for each assembly and analysis of contigs)
---- scaffolds (contains links to scaffolds for each assembly and analysis of scaffolds)
--- ...
```

The directory structure is created as the assemblers run. So the full file structure may not be visible straight after MASS starts. Also, we create the symbolic links to unitigs, contigs and scaffolds on an as needed basis. Some assemblers may not produce certain types of assembled sequences and in those cases we do not create the associated links directory.

Troubleshooting

Here are some issues that you might run into during the MASS stage:

1. ABySS installed but without MPI support. RAMPART requires ABySS to be configured with openmpi in order to use parallelisation in ABySS. If you encounter the following error message lease reinstall ABySS and specify the –with-mpi option during configuration:

```
mpirun was unable to find the specified executable file, and therefore did not launch the job.  This
reported for process rank 0; it may have occurred for other processes as well.

NOTE: A common cause for this error is misspelling a mpirun command
    line parameter option (remember that mpirun interprets the first
    unrecognized command line token as the executable).
```

5.3.4 Analyse assemblies

RAMPART currently offers 3 assembly analysis options:

- Contiguity
- Kmer read-assembly comparison
- Completeness

These types of analyses can be executed in either the `analyse_mass` or `analyse_amp` pipeline element. The available tool options for the analyses are: QUAST,KAT,CEGMA.

QUAST, compares the assemblies from a contiguity perspective. This tool runs really fast, and produces statistics such as the N50, assembly size, max sequence length. It also produces a nice html report showing cumulative length distribution curves for each assembly and GC content curves.

KAT, performs a kmer count on the assembly using Jellyfish, and, assuming kmer counting was requested on the reads previously, will use the Kmer Analysis Toolkit (KAT) to create a comparison matrix comparing kmer counts in the reads to the assembly. This can be visualised later using KAT to show how much of the content in the reads has been assembled and how repetitive the assembly is. Repetition could be due to heterozygosity in the diploid genomes so please read the KAT manual and walkthrough guide to get a better understanding of how to interpret this data. Note that information for KAT is not automatically used for selecting the best assembly at present. See next section for more information about automatic assembly selection.

CEGMA aligns highly conserved eukaryotic genes to the assembly. CEGMA produces a statistic which represents an estimate of gene completeness in the assembly. i.e. if we see CEGMA maps 95% of the conserved genes to the assembly we can assume that the assembly is approximately 95% complete. This is a very rough guide and shouldn't be taken literally, but can be useful when comparing other assemblies made from the same data. CEGMA has a couple of other disadvantages however, first it is quite slow, second it only works on eukaryotic organisms so is useless for bacteria.

An example snippet for a simple and fast contiguity based analyses is as follows:

```
<analyse_mass>
   <tool name="QUAST" threads="16" memory="4000"/>
</analyse_mass>
```

In fact we strongly recommend you use QUAST for all your analyses. Most of RAMPART's system for scoring assemblies (see below) is derived from Quast metrics and it also runs really fast. The runtime is insignificant when compared to the time taken to create assemblies. For more a complete analysis, which will take a significant amount of time for each assembly, you can request KAT and CEGMA. An example showing the use of all analysis tools is as follows:

```
<analyse_mass parallel="false">
   <tool name="QUAST" threads="16" memory="4000"/>
   <tool name="KAT" threads="16" memory="50000" parallel="true"/>
   <tool name="CEGMA" threads="16" memory="20000"/>
</analyse_mass>
```

Note that you can apply `parallel` attributes to both the `analyse_mass` and individual tool elements. This enables you to select those process to be run in parallel where possible. Setting `parallel="true"` for the `analyse_mass` element will override `parallel` attribute values for specific tools.

Selecting the best assembly

Assuming at least one analysis option is selected, RAMPART will produce a summary file and a tab separated value file listing metrics for each assembly, along with scores relating to the contiguity, conservation and problem metrics, and a final overall score for each assembly. Each score is given a value between 0.0 and 1.0, where higher values represent better assemblies. The assembly with the highest score is then automatically selected as the **best** assembly to be used downstream. The group scores and the final scores are derived from underlying metrics and can be adjusted to have different weightings applied to them. This is done by specifying a weighting file to use in the RAMPART pipeline.

By default RAMPART applies its own weightings, which can be found at `<rampart_dir>/etc/weightings.tab`, so to run the assembly selection stage with default settings the user simply needs add the following element to the pipeline:

```
<select_mass/>
```

Should the user wish to override the default weights that are assigned to each assembly metric, they can do so by setting the `weightings_file` attribute element. For example, using an absolute path to a custom weightings file the XML snippet may look like this:

```
<select_mass weightings_file="~/.tgac/rampart/custom_weightings.tab"/>
```

The format of the weightings key value pair file separated by '=' character. Comment lines can start using '#'. Most metrics are derived from Quast results, except for the core eukaryote genes detection score which is gathered from CEGMA. Note, that some metrics from Quast will only be used in certain circumstances. For example, the na50 and nb_ma_ref metrics are only used if a reference is supplied in the organism element of the configuration file. Additionally, the nb_bases, nb_bases_gt_1k and the gc% metrics are used only if the user has supplied either a reference, or has provided estimated size and / or estimated gc% for the organism respectively.

TODO: Currently the kmer metric, is not included. In the future this will offer an alternate means of assessing the assembly completeness.

The file best.fa is particularly important as this is the assembly that will be taken forward to the second half of the pipeline (from the AMP stage). Although we have found that scoring system to be generally quite useful, we strongly recommend users to make their own assessment as to which assembly to take forward as we acknowledge that the scoring system is biased by outlier assemblies. For example, consider three assemblies with an N50 of 1000, 1100 and 1200 bp, with scaled scores of 0, 0.5 and 1. We add a third assembly, which does poorly and is disregarded by the user, with an N50 of 200 bp. Now the weighted N50 scores of the assemblies are 0, 0.8, 0.9 and 1. Even though the user has no intention of using that poor assembly, the effective weight of the N50 metric of the three good assemblies has decreased drastically by a factor of (1 - 0) / (1 - 0.8) = 5. It's possible that the assembly selected as the best would change by adding an irrelevant assembly. For example consider two metrics, a and b, with even weights of 0.5 for three assemblies, and then again for four assemblies after adding a fourth irrelevant assembly, which performs worst in both metrics. By adding a fourth irrelevant assembly, the choice of the best assembly has changed.

Three assemblies: a = {1000, 1100, 1200}, b = {0, 10, 8} sa = {0, 0.5, 1}, sb = {0, 1, 0.8} fa = {0, 0.75, 0.9} best = 0.9, the assembly with a = 1200

Four assemblies: a = {200, 1000, 1100, 1200}, b = {0, 0, 10, 8} sa = {0, 0.8, 0.9, 1}, sb = {0, 0, 1, 0.8} fa = {0, 0.4, 0.95, 0.9} best = 0.95, the assembly with a = 1100

To reiterate, we recommend that the user double check the results provided by RAMPART and if necessary overrule the choice of assembly selected for further processing. This can be done, i.e. starting from the AMP stage with a user selected assembly, by using the following command: `rampart -2 -a <path_to_assembly> <path_to_job_config>`.

Analysing assemblies produced by AMP

In addition, to analysing assemblies produced by the MASS stage, the same set of anlayses can also be applied to assemblies produced by the AMP stage. The same set of attributes that can be applied to `analyse_mass` can be applied to `analyse_amp`. In addition, it is also possible to specify an additional attribute: `analyse_all`, which instructs RAMPART to analyse

> assemblies produced at every stage of the AMP pipeline. By default only the final assembly is analysed. Also note that there is no need to select assemblies from AMP, so there is no corresponding `select_amp` element.

5.3.5 AMP - Assembly Improver

This stage takes a single assembly as input and tries to improve it. For example, additional scaffolding, gap filling can be performed at this stage. Currently, AMP supports the following tools:

- Platanus_Scaffold_V1.2
- SSPACE_Basic_V2.0
- SOAP_Scaffold_V2.4
- Platanus_Gapclose_V1.2
- SOAP_GapCloser_V1.12
- Reapr_V1

AMP stages accept `threads` and `memory` attributes just like MASS and MECQ. A simple XML snippet describing a scaffolding and gap closing process is shown below:

```
<amp>
   <stage tool="SSPACE_Basic_V2.0" threads="8" memory="16000">
      <inputs>
         <input name="mp1" ecq="raw"/>
      </inputs>
   </stage>
   <stage tool="SOAP_GapCloser_V1.12" threads="6" memory="8000">
      <inputs>
         <input name="pe1" ecq="raw"/>
      </inputs>
   </stage>
</amp>
```

Each stage in the AMP pipeline must necessarily run linearly as each stage requires the output from the previous stage. The user can specify additional arguments to each tool by adding the `checked_args` attribute to each stage. For example to specify that SSPACE should use PE reads to extend into gaps and to cap min contig length to scaffold as 1KB:

```
<amp>
   <stage tool="SSPACE_Basic_V2.0" threads="16" memory="32000" checked_args="-x 1 -z 1000">
      <inputs>
         <input name="mp1" ecq="raw"/>
      </inputs>
   </stage>
   <stage tool="SOAP_GapCloser_V1.12" threads="6" memory="8000">
      <inputs>
         <input name="pe1" ecq="raw"/>
      </inputs>
   </stage>
</amp>
```

Output from amp will be placed in a directory called `amp` within the job's output directory. Output from each stage will be placed in a sub-directory within this and a link will be created to the final assembly produced at the end of the amp pipeline. This assembly will be used in the next stage.

Some assembly enhancement tools such as Platanus scaffolder, can make use of bubble files in certain situations to provide better scaffolding performance. When you are assembling a diploid organism (i.e. the ploidy attribute of your organism element is set to "2") and the assembler used in the MASS step produces bubble files, then these are automatically passed onto the relevant AMP stage.

Some assembly enhancement tools require the input contigs file to have the fasta headers formatted in a particular way, and sometimes with special information embedded within it. For example, SOAP scaffolder cannot process input files that contain gaps, and the platanus scaffolder must contain the kmer coverage value of the contig in the header. Where possible RAMPART tries to automatically reformat these files so they are suitable for the assembly enhancement tool. However, not all permutations are catered for, and some combinations are probably not possible. If you are aware of any compatibility issues between contig assemblers and assembly enhancement tools that RAMPART is not currently addressing correctly, then please raise a ticket on the RAMPART github page: https://github.com/TGAC/RAMPART/issues, with details and we will try to fix the issue in a future version.

In the future we plan to make the AMP stage more flexible so that it can handle parameter optimisation like the MASS stage.

Troubleshooting

If you encounter a error message relating to "finalFusion" not being available during inital checks it is likely that you nneed to find the "finalFusion" executable and install this along with SOAP scaffolder. This can be difficult to find, and note this is not the same as SOAPFusion. If you can not find this online, please contact daniel.mapleson@tgac.ac.uk.

5.3.6 Finaliser

The final step in the RAMPART pipeline is to finalise the assembly. Currently, this simply involves standardising the assembly filename and headers in the fasta file. The finaliser can be invoked this like to use the specifed prefix:

```
<finalise prefix="E.coli_Sample1_V1.0"/>
```

If a prefix is not specified RAMPART will build it's own prefix based on the information provided in the job configuration file, particularly from the organism details.

The finalising process will take scaffolds and artifically break them into contigs where there are large gaps present. The size of the gap required to trigger a break into contigs is defined by the `min_n` attribute. By default this is set to 10. An example, that applies a prefix and breaks to contigs only on gaps larger than size 20 is shown below:

```
<finalise prefix="E.coli_Sample1_V1.0" min_n="20"/>
```

The input from this stage will either be the best assembly selected from MASS, or the final assembly produced by AMP depending on how you've setup your job. The output from this stage will be as follows:

- `<prefix>.scaffolds.fa` (the final assembly which can be used for annotation and downstream analysis)
- `<prefix>.contigs.fa` (the final set of scaffolds are broken up where stretches of N's exceed a certain limit)
- `<prefix>.agp` (a description of where the contigs fit into the scaffolds)
- `<prefix>.translate` (how the fasta header names translate back to the input assembly)

All the output files should be compressed into a single tarball by default. You can turn this functionality off by adding the `compress="false"` attribute to finaliser.

5.4 Potential runtime problems

There are a few issues that can occur during execution of RAMPART which may prevent your jobs from completing successfully. This part of the documentation attempts to list common problems and suggests workarounds or solutions:

- Quake fails

In this case, if you have set the quake k value high you should try reducing it, probably to the default value unless you know what you are doing. Also Quake can only work successfully if you have sufficient sequencing depth in your dataset. If this is not the case then you should either obtain a new dataset or remove quake error correction from your RAMPART configuration and try again.

- Kmergenie fails

Often this occurs for the same reasons as Quake, i.e. inadequate coverage. Check that you have correct set the ploidy value for your organism in the configuration file (Kmer genie only support haploid or diploid (i.e. 1 or 2), for polyploid genomes you are on your own!) Also keep in mind that should you remove kmer genie from your pipeline and manually set a kmer value for an assembler, it is unlikely that your assembly will be very contiguous but RAMPART allows you to try things out and you maybe able to assemble some useful data.

- Pipeline failed at a random point during execution of one of the external tools

In this case check your system. Ensure that the computing systems are all up and running, that there have been no power outages and you have plenty of spare disk space. RAMPART can produce a lot of data for non-trivial genomes so please you have plenty of spare disk space before starting a job.

CHAPTER 6

Multi-sample projects

Due to the falling cost and increasing capacity of sequencing devices, as well as improvements in the automation of library preparation, it is now possible to sequence many strains of the same species. Assembling these strains in isolation can be time consuming and error prone, even with pipelines such as RAMPART. We therefore provide some additional tools to help the bioinformatician manage these projects and produce reasonable results in short time frames.

The logic we use to approach multi-sample projects is as follows:

1. Use jellyswarm to assess all samples based on their distinct kmer count after rare kmers (sequencing errors) are excluded. If there is little agreement between samples then more analysis is required. If there is good agreement then carry on to 2.
2. Exclude outlier samples. These must be assembled and analysed separately.
3. Use RAMPART to help develop an assembly recipe for a few of the more typical strains (strains where distinct kmer count is close to the mean) to attempt to identify optimal parameters, particularly the k parameter. If there is little agreement in parameters between samples then all strains must be looked at in isolation. If there is good agreement carry on to 4.
4. Use RAMPART in multi-sample configuration to execute the recipe for all strains.

Note: This is by no means the only way to approach these projects, nor will it necessarily give the best results, but it should allow a single bioinformatician to produce reasonable assemblies for a project with hundreds or thousands of samples within a reasonable timeframe given appropriate computational resources.

6.1 Jellyswarm

Jellyswarm uses the jellyfish K-mer counting program to count K-mers found across multiple samples. Jellyswarm will attempt to exclude K-mers that are the result of sequencing errors from the results. It then analyses the number of distinct k-mers found across all samples and records basic statistics such as the mean and standard deviation of the distribution.

The syntax for running Jellyswarm from the command line is: `jellyswarm [options] <directory_containing_samples>`. To get a list and description of available options type: `jellyswarm --help`. Upon starting jellyswarm will search for an environment configuration file and a logging configuration exactly like RAMPART. Jellyswarm will then configure the execution environment as appropriate and then run the pipeline.

Jellyswarm finds fastq samples associated with the samples by interrogating a directory containing all the files. It then sorts by name the files found in that directory with an ".fq" or ".fastq" extension. By default we assume paired end sequencing was used and group each pair of fastq files together. If you have interleaved fastq files or have single

end data then activate the single end mode by using the `-1` command line option. Jellyswarm can also interogate all subdirectories in the parent directory for fastq files by using the `-r` command line option.

You can control the amount of resources jellyswarm uses on your HPC environment by using a few other options. The `-m` option allows you to specify a memory limit for each jellyfish count instance that is run. The amount of memory required will be determined by the size of your hash and the size of the genome in question. Depending on the particular environment used this limit may either represent a hard limit, i.e. if exceeded the job will fail (this is the case on PBS), or it may represent a resource reservation where by this amount of memory is reserved for the running job (this is the case on LSF). For LSF, your job may or may not fail if the memory limit is exceeded depending on the availability of memory on the node on which your job is running. The number of threads per process is controlled using the `-t` option. Finally, you can k-mer count all samples in parallel by using the

6.2 RAMPART multi-sample mode

RAMPART can be executed in multi-sample mode by removing the ‘‘libraries” element from the configuration file and replacin
it with a ‘‘samples” element containing a ‘‘file” attribute describing the path to a file containing a list of sample libraries to process. For example:

```
    <samples file="reads.lst"/>
```

The file containing the sample libraries should be a tab separated file with columns describing the following:

1. Sample name
2. Phred
3. Path to R1 file
4. Path to R2 file

For example:

```
PRO461_S10_B20  PHRED_33        S10_B20_R1.fastq  S10_B20_R2.fastq
PRO461_S10_D20  PHRED_33        S10_D20_R1.fastq  S10_D20_R2.fastq
PRO461_S10_F20  PHRED_33        S10_F20_R1.fastq  S10_F20_R2.fastq
PRO461_S11_H2   PHRED_33        S11_H2_R1.fastq   S11_H2_R2.fastq
```

We may extend this format to include additional columns describing library options in the future.

In addition to replacing the libraries element with the samples element, you should also add a ‘‘collect” element inside
the ‘‘pipeline” element:

```
    <collect threads="2" memory="5000"/>
```

A complete multi-sample configuration file might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<rampart author="dan" collaborator="someone" institution="tgac" title="Set of C.Coli assemblies">
      <organism name="ccoli_nextera" ploidy="1">
              <reference name="C.Coli_15-537360" path="Campylobacter_coli_15_537360.GCA_000494775.1.2
      </organism>
      <samples file="reads.lst"/>
      <pipeline parallel="true">
              <mecq>
                      <ecq name="tg" tool="TrimGalore_V0.4" threads="2" memory="5000"/>
              </mecq>
              <mass>
                      <job name="spades_auto" tool="Spades_V3.1" threads="2" memory="10000">
                              <kmer min="71" max="111" step="COARSE"/>
```

```
                                <coverage list="50"/>
                                <inputs>
                                        <input ecq="tg"/>
                                </inputs>
                        </job>
                </mass>
                <mass_analysis>
                        <tool name="QUAST" threads="2" memory="5000"/>
                </mass_analysis>
                <mass_select threads="2" memory="5000"/>
                <finalise prefix="Ccoli_Nextera"/>
                <collect threads="2" memory="5000"/>
        </pipeline>
</rampart>
```

You can then start RAMPART in the normal way. RAMPART will output the stages directories as normal but as subdirectories within a sample directory.

Currently, there are also a number of supplementary scripts to aid the analysis of data across all samples and annotation of each sample using PROKKA (http://www.vicbioinformatics.com/software.prokka.shtml). Note that PROKKA is only relevant for prokaryotic genomes. The scripts were designed for execution on LSF environments, so some modification of the scripts may be necessary should you wish to execute in other scheduled environments or on unscheduled systems. Whilst each script comes with its own help message and man page, we so not provide extensive documentation for these and leave it to the bioinformatician to tweak or reuse the scripts as they see fit. We plan to incorperate a mechanism into RAMPART to enable it to properly handle prokaryotic genome annotation in the future.

CHAPTER 7

Citing RAMPART

To cite RAMPART please use the following reference:

RAMPART: a workflow management system for de novo genome assembly Daniel Mapleson; Nizar Drou; David Swarbreck Bioinformatics 2015; doi: 10.1093/bioinformatics/btv056

Additional information:

PMID: 25637556 Github page: https://github.com/TGAC/RAMPART Project page: http://www.tgac.ac.uk/rampart/ Latest manual: http://rampart.readthedocs.org/en/latest/index.html

CHAPTER 8

License

RAMPART is available under GNU GLP V3: http://www.gnu.org/licenses/gpl.txt

For licensing details of other RAMPART dependencies please consult their own documentation.

CHAPTER 9

Contact

Daniel Mapleson - Analysis Pipelines Project Leader at The Genome Analysis Centre (TGAC)

Website: http://www.tgac.ac.uk/bioinformatics/genome-analysis/daniel-mapleson/

Email: daniel.mapleson@tgac.ac.uk

CHAPTER 10

Acknowledgements

- Nizar Drou
- David Swarbreck
- Bernardo Clavijo
- Robert Davey
- Sarah Bastkowski
- Tony Burdett
- Ricardo Ramirez
- Purnima Pachori
- Mark McCullen
- Hugo Taveres
- Ram Krishna Shrestha
- Darren Waite
- Tim Stitt
- Shaun Jackman
- And everyone who contributed to making the tools RAMPART depends on!