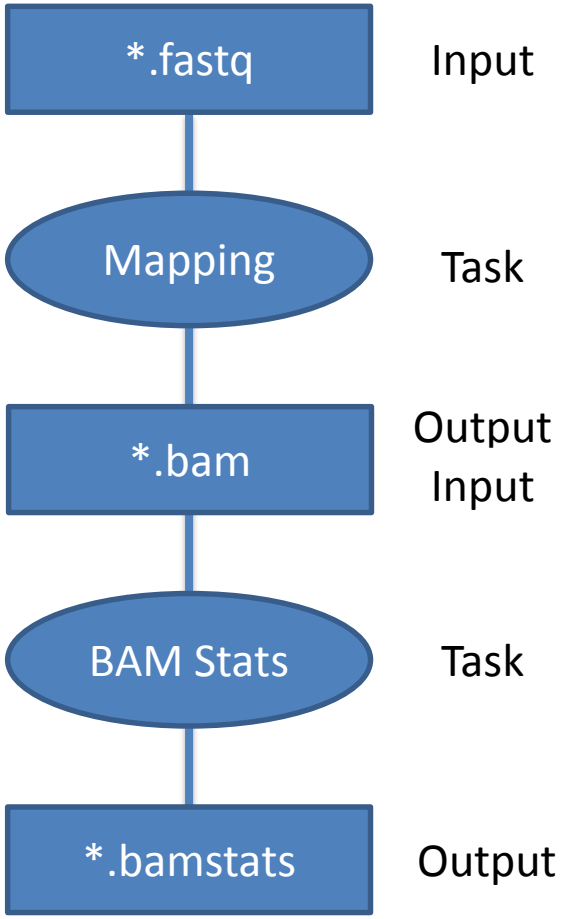
Writing reproducible workflows

Oxford Biomedical Data Science Training Programme

Workflows

- Perform multiple processing steps in turn
- Start a task only when all upstream dependencies are completed
- Perform steps in parallel for multiple input files



Workflow Tools

- Pipeline scripts (Bash, Python…)
- Makefiles
- Graphical User Interface workbenches
  - Taverna
  - Galaxy
  - Genomatix, CLCBio, Partek, …
- Programmatic workflow systems
  - Nextflow
  - Snakemake
  - Ruffus / CGAT-core

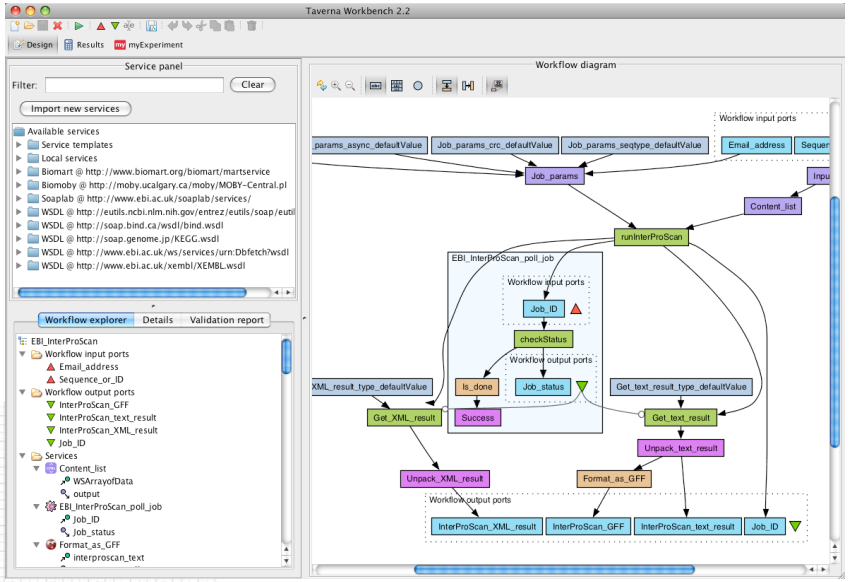
See also: https://github.com/pditommaso/awesome-pipeline

Bioinformatics Workbenches

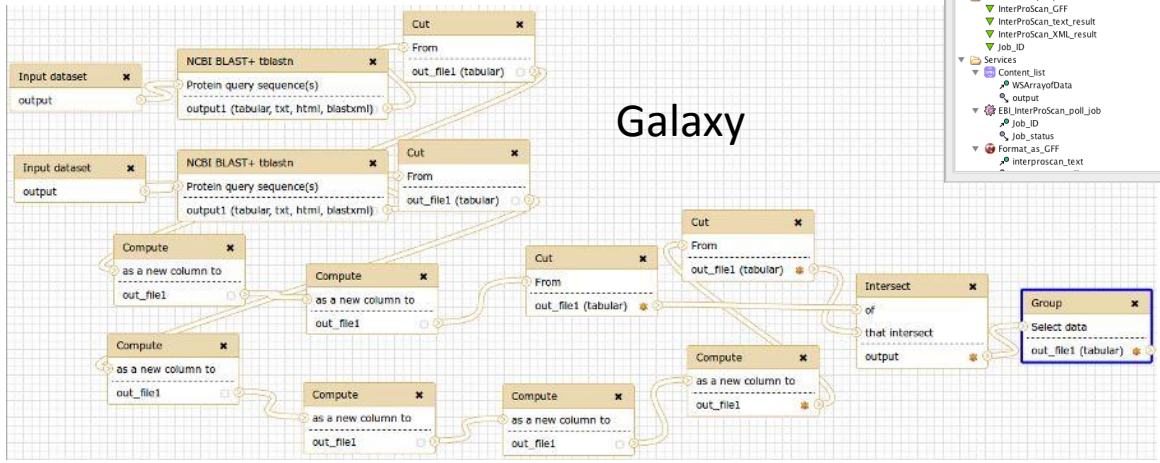
Advantages
No programming required

Disadvantages
Black-box
Limited to available tools
Limited complexity
Require expert administration

Taverna



Galaxy

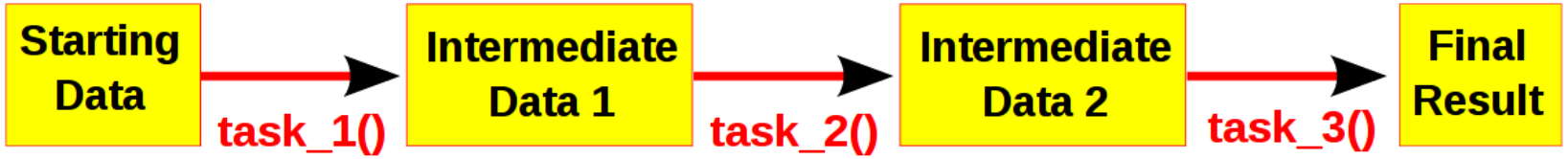


Commercial systems:
Genomatix, CLCBio, Partek,

Ruffus

- Open-source workflow library for Python
- **Lightweight**: Suitable for the simplest of tasks
- **Scalable**: Handles even fiendishly complicated pipelines
- **Standard python**: Uses decorators to wrap standard functions

- Flow control from one function to the next
- Allows tasks to be completed sequentially
- If the pipeline fails, ruffus will restart where it left off



http://www.ruffus.org.uk/

CGATcore

- Wrapper for Ruffus
- Flexible parameterisation
- Detailed logging
- Database interaction
- Interaction with HPC clusters - DRMAA
- Simplified pipeline control

Building & running Ruffus/cgat-core pipelines

1. Write pipeline script
   - Series of Python functions (tasks)
   - Import Ruffus to manage task dependencies (decorators)
   - Import CGATcore to manage parametrisation, logging, cluster and database interaction and pipeline running
2. Write configuration files
   - Define all pipeline parameters in pipeline.yml
3. Run from the command line
4. Examine output
   - Log files
   - Output files
   - Database tables
   - Reports with plots (Rmarkdown, Notebooks)

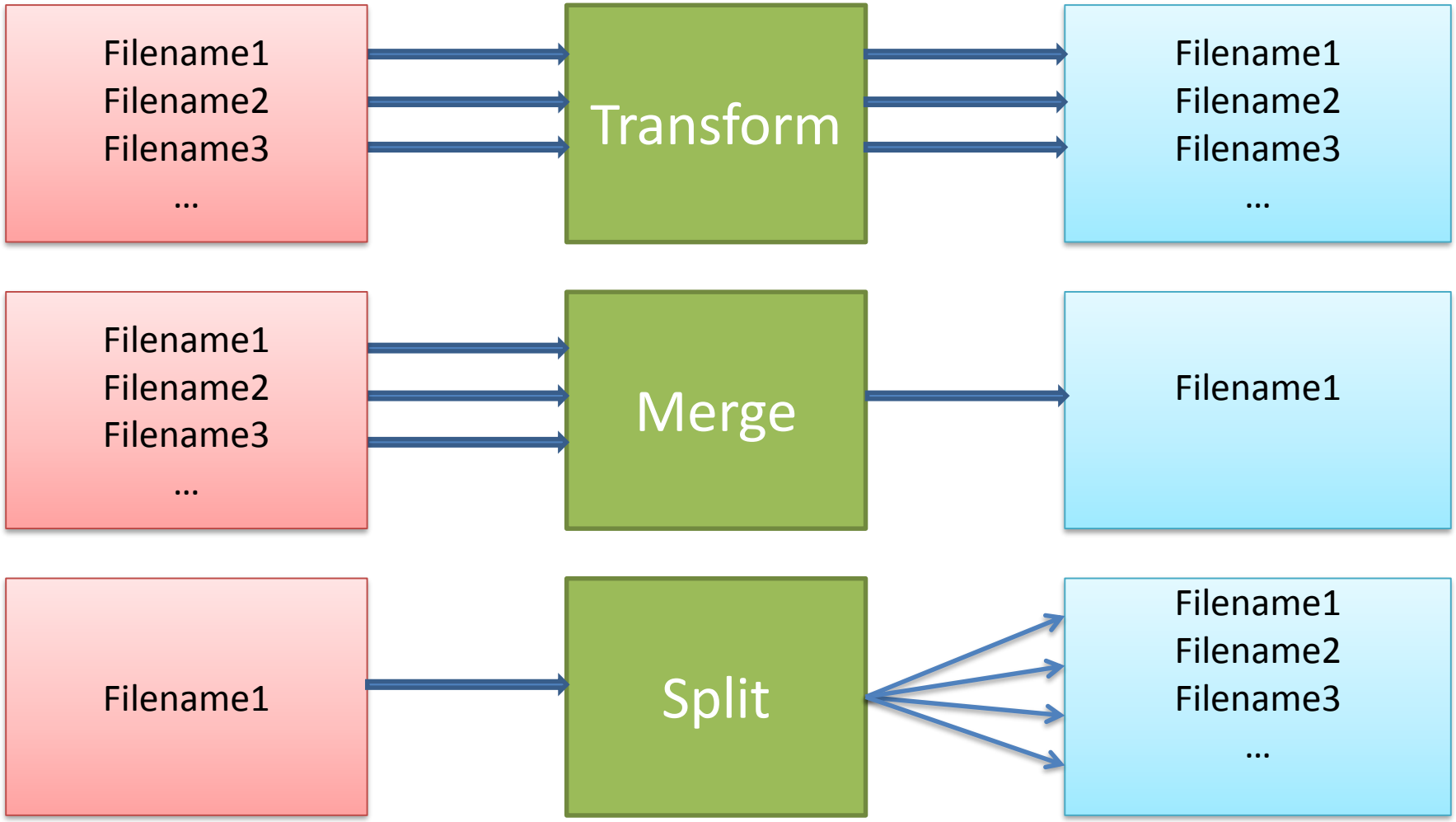
Ruffus Tasks

Filename1
Filename2
….

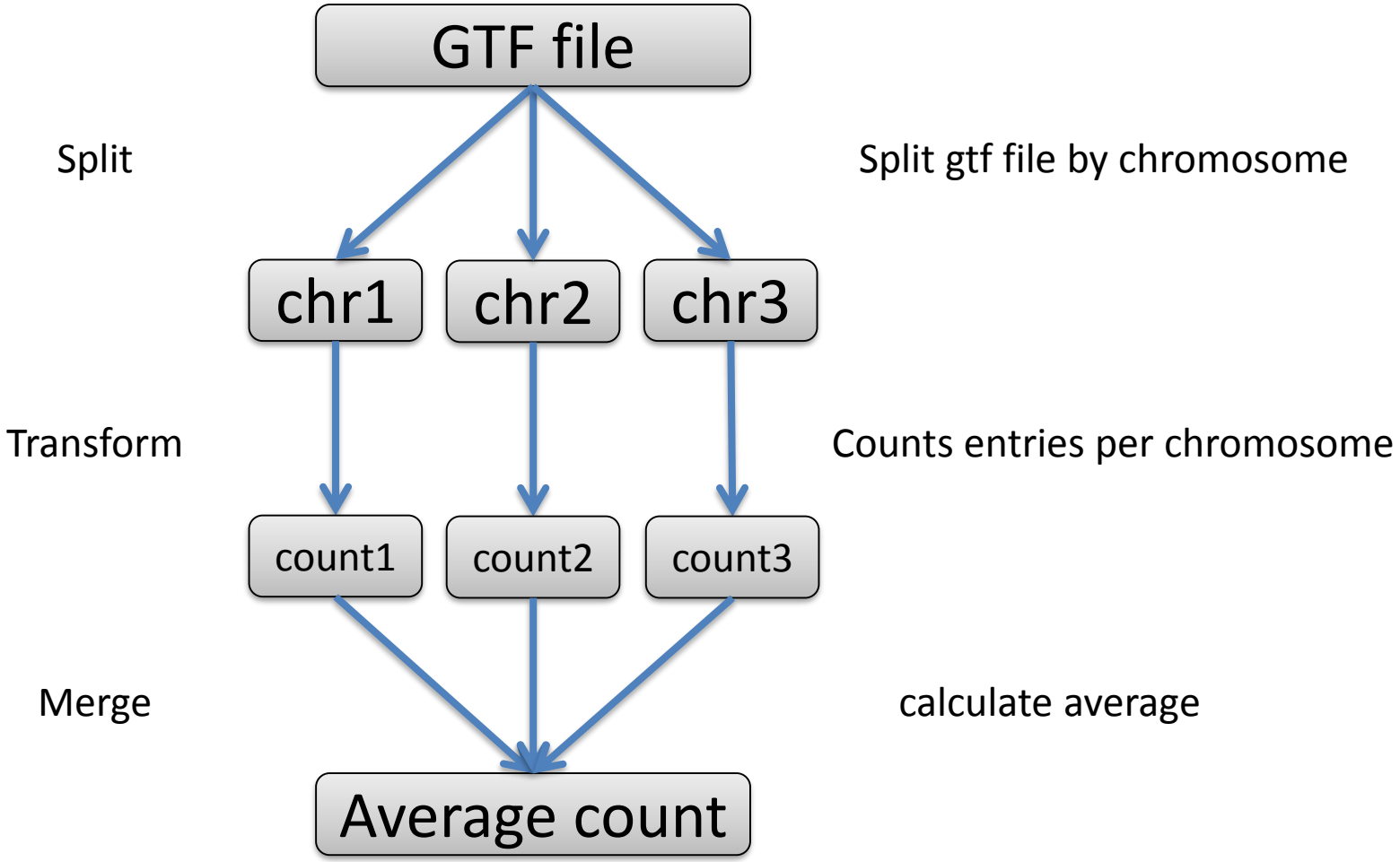
Python function

Filename1
Filename2
….

Basic operations



Example pipeline



Defining Tasks / functions

Split

```
def split_chrom(infile, outfiles):
    with open(infile) as inf:
        for line in inf:
```

Any Python code

Transform

```
def count_genes(infile, outfile):
    statement = '''wc -l %(infile)
                   > %(outfile)'''
    P.run(statement)
```

Any command line statement

Merge

```
def average(infiles, outfile):
    for infile in infiles:
        with open(infile) as inf:
            count = inf.read()
```

Python Decorators

- Decorators dynamically alter the functionality of a function
- Enables you to run a function before another function call
- @split
  - Checks dependencies are satisfied
  - configures input & output files
  - passes them to inner function

Defining Task Input & Output

Split

```
@split('all.gtf', 'chr*.gtf')
def split_chrom(infile, outfiles):
```

Glob or regex for output file name

Transform

```
@transform( [ 'chr1.gtf', 'chr2.gtf'],
                                suffix('.gtf'),
                                '.counts')
def count_genes(infile, outfile):
```

List of input files

Regular expression

Output file regex

Merge

```
@merge(['chr1.counts','chr2.counts'],
                                'all.average')
def average( infiles, outfile ):
```

List of input files

Output file

Combining Tasks

Split

```
@split('all.gtf', 'chr*.gtf')
def split_chrom(infile, outfiles):
```

Transform

```
@transform( [ 'chr1.gtf', 'chr2.gtf'],
                                suffix('.gtf'), '.counts')
def count_genes(infile, outfile):
```

Merge

```
@merge(['chr1.counts','chr2.counts'],
                                'all.average')
def average(infiles, outfile):
```

Combining Tasks

Split

```
@split('all.gtf', 'chr*.gtf')
def split_chrom(infile, outfiles):
```

Transform

```
@transform(split_chrom, suffix('.gtf'), '.counts')
def count_genes(infile, outfile):
```

Merge

```
@merge(count_genes, 'all.average')
def average(infiles, outfile):
```

General pipeline structure

Documentation

Description of pipeline and dependencies

Import section

```
import ruffus
from cgatcore  import pipeline as P
```

Read Parameters

```
Params = P.get_parameters("pipeline.yml")
```

Load options from config file

Parameters added to PARAMS dictionary

Define Tasks

Write processing functions and combine

__main__

```
if __name__ == "__main__":
    sys.exit( P.main(sys.argv) )
```

If script run from command line
Run main function from pipeline.py

Parameter Definition

pipeline.yml

Usually located in pipeline working directory

```
section:
    key: value
```

indentation

File organised into sections

Add parameters as key-value pairs
Prefixes section name to key

```
genome: hg19

bowtie:
    index_dir: /databank/bowtie
```

```
params =
{
  'genome' : 'hg19',
  'bowtie_index_dir':  '/databank/bowtie'
}
```

Parameters added to params dictionary

Inside a task

glob

Specify input files
Regular expression

Insert regex group here

```
@transform( '*.fastq.gz', regex(r'(.*).fastq.gz'),  r'trim/\1.fastq.gz')
```

Raw string literal
(no escape characters)

Match any character
0 or more time
Extract as group

```
def trim_reads(infile, outfile):
        statement = '''
                zcat %(infile)s
                | fastx_trimmer %(trim_options)s
                2> %(outfile)s.log
                | gzip > %(outfile)s'''
        P.run(statement, job_queue = 'longjobs.q',
                job_threads = 8, job_memory = '8G'
                job_condaenv = 'obds_py3')
```

Variable
interpolation

Parameter
interpolation

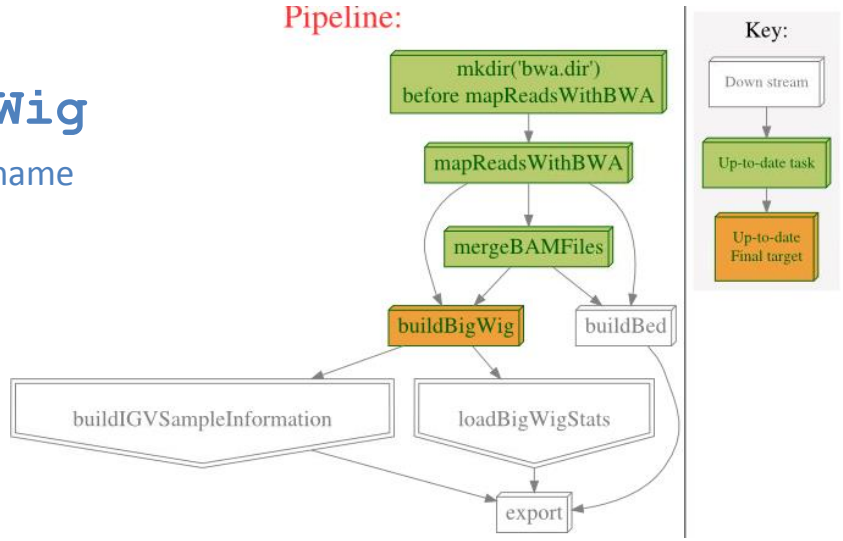
CGATcore command

Running from the command line

- CGATcore enables us to interact with Ruffus pipelines using ***actions***
  - **show** – show which functions will run when you run the pipeline
  - **plot** – plot a schematic showing the status of the pipeline functions
  - **make** – run the pipeline
  - **config** – create default configuration files for the pipeline

```
$ python mapping.py plot buildBigWig
```

Pipeline script Action Target function name



Running from the command line

- To run a task append the name of the function you want to run to the “make” action
- Ruffus will run everything up to and including this function if:
  - output is not already present
  - Timestamp of input file is more recent than timestamp of output file

```
$ python pipeline_test.py make split_chrom
```

- Command line options:
  - v verbosity level (recommend 5)
  - Controls level of logging messages printed to screen / log file

```
$ python pipeline_test.py make split_chrom -v 5
```

Grouping tasks

- Pipelines can have many independent branches
- It is often helpful to add dummy functions
  - after each section of the pipeline
  - "full" at the end to run the whole pipeline (convention)

```
@follows(loadReadCounts,  loadPicardStats,
                 loadBAMStats,  loadContextStats)
def general_qc():
   pass

$ python mapping.py make general_qc –v 5
```

Enables you to run all of the QC sections

```
@follows(mapping, qc, views, duplication)
def full():
   pass

$ python mapping.py make full –v 5
```

Enables you to run all four sections of the pipeline

Pipeline Status

- Keep track of which functions have run:
  - Presence of files
  - Time-stamps of files
  - Logging database (not currently used)

Exercise 1

- Write a split-transform-merge pipeline
- Split file by chromosome
- Count the number of transcripts for each chromosome
- Read all count files, calculate the average and write to file