BIOINFORMATICS APPLICATIONS NOTE Vol. 26 no. 21 2010, pages 2778–2779
doi:10.1093/bioinformatics/btq524

Genome analysis Advance Access publication September 16, 2010

Ruffus: a lightweight Python library for computational pipelines

Leo Goodstadt

Medical Research Council Functional Genomics Unit, Department of Physiology, Anatomy and Genetics, University of Oxford, Oxford OX1 3QX, UK

Associate Editor: Martin Bishop

ABSTRACT

Summary: Computational pipelines are common place in scientific research. However, most of the resources for constructing pipelines are heavyweight systems with graphical user interfaces. Ruffus is a library for the creation of computational pipelines. Its lightweight and unobtrusive design recommends it for use even for the most trivial of analyses. At the same time, it is powerful enough to have been used for complex workflows involving more than 50 interdependent stages.

Availability and implementation: Ruffus is written in python. Source code, a short tutorial, examples and a comprehensive user manual are freely available at http://www.ruffus.org.uk. The example program is available at http://www.ruffus.org.uk/examples/bioinformatics

Contact: ruffus@llew.org.uk

Received on August 4, 2010; revised on August 31, 2010; accepted on September 9, 2010

1 INTRODUCTION

Large-scale computational analyses are now integral to many biological studies. 'Workflow' management systems have accordingly proliferated, including Taverna (Oinn *et al*., 2004), Biopipe (Hoon *et al*., 2003) and Pegasys (Shah *et al*., 2004). These are highly featured, designed for automated and robust operation even by non-expert users, managed using graphics user interfaces and specified in XML or proprietary domain-specific languages.

However, these workflow systems can be too cumbersome for explorative and empirical studies with novel datasets. The appropriate scientific approach cannot always be determined a priori. On the other hand, the advantages of computational pipelines over *ad hoc* scripts, even for simple tasks, are all more apparent with increasingly complex datasets and the use of parallel processing.

The standard Unix build (software construction) system 'make' has been widely used to keep track of dependencies in scientific pipelines. 'Makefiles' specify the files names of data for the input and output of each stage of a pipeline as well as the 'rules' (commands) for generating each type of output from its corresponding input. The entire pipeline is represented by a statically inferred dependency (directed acyclic) graph for the succession of data files. The same 'rule' can be applied to multiple data files at the same time, for example, to run BLAST searches on many sequence files in parallel. Automatic data tracking in pipelines allows only the out-of-date parts of the analyses to be rescheduled and recalculated, with minimal redundancy. This is necessary when parts of the pipeline are subject to rapid cycles of development or where the underlying data is being generated continually.

Unfortunately, 'make' is not a good fit for the design of scientific pipelines. 'Make' specifications are written in an obscure and limited language. (This is mitigated in 'make' replacements such as 'scons' or Ruby 'rake'). Pipeline dependencies are not specified directly but inferred by the 'make' program by linking together 'rules' in the right order. This means that scientific pipelines can be difficult to develop, understand and debug.

So-called 'embarrassingly parallel' problems are particularly common in bioinformatics; examples include BLAST and HMMer searches of sequence databases, or region-by-region genome annotation. The number of parallel operations needed varies at 'run-time' with the presented data: a larger sequence file might be split up into smaller fragments to be processed in parallel. However, 'make' systems and their kin require all operations in a pipeline to be determined when the build script is analysed, because of the reliance on static, pre-calculated dependency graphs. They cannot easily deal with, for example, the splitting up of large problems into smaller fragments to be computed in parallel, if the number of such fragments depends on the input data and runtime conditions, and can only be determined in the middle of running the pipeline.

In this article, we present a new lightweight library for computational pipelines that explicitly supports these programming tasks. Some of its main advantages of Ruffus are:

- Ruffus configuration files are normal Python scripts. Python is a modern dynamically typed programming language known for its elegance, simplicity, and that is already widely used in the bioinformatics community (Cock *et al*., 2009). Standard Python tools can be used to develop and debug Ruffus scripts.
- Like 'makefiles', Ruffus scripts can run only the out-of-date parts of the pipeline, using parallel processing if appropriate.
- Pipeline dependencies are specified explicitly for maximal clarity and ease of documentation.
- A flowchart of the pipeline can be printed out in a variety of graphical formats. Detailed trace output is available, documenting which operations are up-to-date or will be run (Fig. 1).

2 DESIGN

Ruffus is a module for the python language that adds lightweight support for computational pipelines. Each stage of the pipeline is a separately written (normal) python function. By convention, strings contained in the first two arguments of pipelined functions are assumed to be names of input and output files for this stage. The modification times of the underlying files are used to determine if this part of the pipeline is up-to-date or not, and should be re-run.

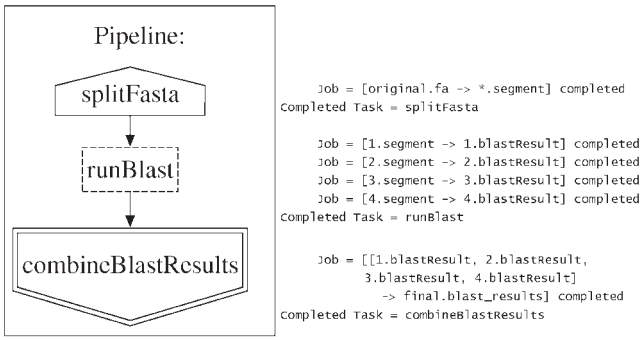

Fig. 1. Trace output and flowchart for a simple Ruffus pipeline.

Table 1. Examples of Ruffus 'decorator' keywords

Ruffus Keyword	Function of annotated pipeline function
Split	Splits up input file into a number of output files (a one-to-many operation)
Transform	Transforms each input into a corresponding output
Merge	Merges multiple input into a single output (a many-to-many operation)
Collate	Group together subsets of input, summarizing each as a separate output.

Ruffus ensures that these pipeline functions are called in the right order with appropriate arguments. For example, when the pipeline specifies BLAST (Altschul *et al*., 1990) searches on four sequence files, three separate calls to the appropriate python function will be made, in parallel if necessary.

To register pipeline stages, Ruffus provides some simple keywords (Table 1) using standard python syntax. These python 'decorators' placed before each function indicates how the stages of the pipeline are linked together, the type of operation and what arguments to supply to each stage of the pipeline.

3 FUNCTION AND EXAMPLES

A standard bioinformatics task for running a blast search efficiently in parallel might involve splitting the initial large sequence file into smaller pieces, calling the BLAST executable for each, and then combining the separate high scoring segment pairs (HSPs) into the final list of matches. These three operations would be represented by three python functions 'decorated' by the '`split`', '`transform`' and '`merge`' Ruffus keywords. The syntax (in outline) would be as follows:

```
from ruffus import *

@split("original.fasta", "*.segment")
def splitFasta(seqFile, segments):
    # code to split sequence file into
    # as many fragments as appropriate
    # depending on the size of "original.fasta"

@transform(splitFasta, suffix(".segment"),
                                ".blastResults")

def runBlast(seqFile, blastResultFile):
     # code to run blast here

@merge(runBlast, "final.blast_results")
def combineBlastResults(blastResultFile,
                        combinedBlastResultFile):
    # code to combine results here

pipeline_run([combineBlastResults], verbose = 3,
                  multiprocess = 5)
```

This will run the three-stage pipeline using up to five processors in parallel, firstly splitting up the starting sequence file '`original.fasta`' into multiple files with the suffix '`.segment`', then running the BLAST program (Altschul *et al*., 1990) to produce corresponding files with the '`.blastResult`' suffix, and finally combining all these into the file '`final.blast_results`'. The trace file for this simple pipeline, as well as its flowchart produced by Ruffus, is shown in Figure 1.

More challenging examples using, for example, the full power of regular expressions to manage pipeline data files, can be found in the Ruffus documentation.

4 CONCLUSION

Ruffus is a python library for programming computational pipelines with lightweight, unobtrusive syntax. It provides all the power of traditional build systems such as automatic data tracking, but in a modern package suited to the needs of bioinformatics. Sample flowcharts of Ruffus pipelines, a tutorial, a detailed manual as well as source code are freely available from http://www.ruffus.org.uk and http://code.google.com/p/ruffus.

ACKNOWLEDGEMENTS

Many thanks to Andreas Heger for advice on the design of Ruffus; Chris Nellåker and T. Grant Belgard for their many suggestions; and Chris Ponting for his support throughout this project.

Funding: Medical Research Council.

Conflict of Interest: none declared.

REFERENCES

Altschul,S.F. *et al*. (1990) Basic local alignment search tool. *J. Mol. Biol*., **215**, 403–410.

Cock,P.J. *et al*. (2009) Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics*, **25**, 1422–1423.

Hoon,S. *et al*. (2003) Biopipe: a flexible framework for protocol-based bioinformatics analysis. *Genome Res*., **13**, 1904–1915.

Oinn,T. *et al*. (2004) Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, **20**, 3045–3054.

Shah,S.P. *et al*. (2004) Pegasys: software for executing and integrating analyses of biological sequences. *BMC Bioinformatics*, **5**, 40.