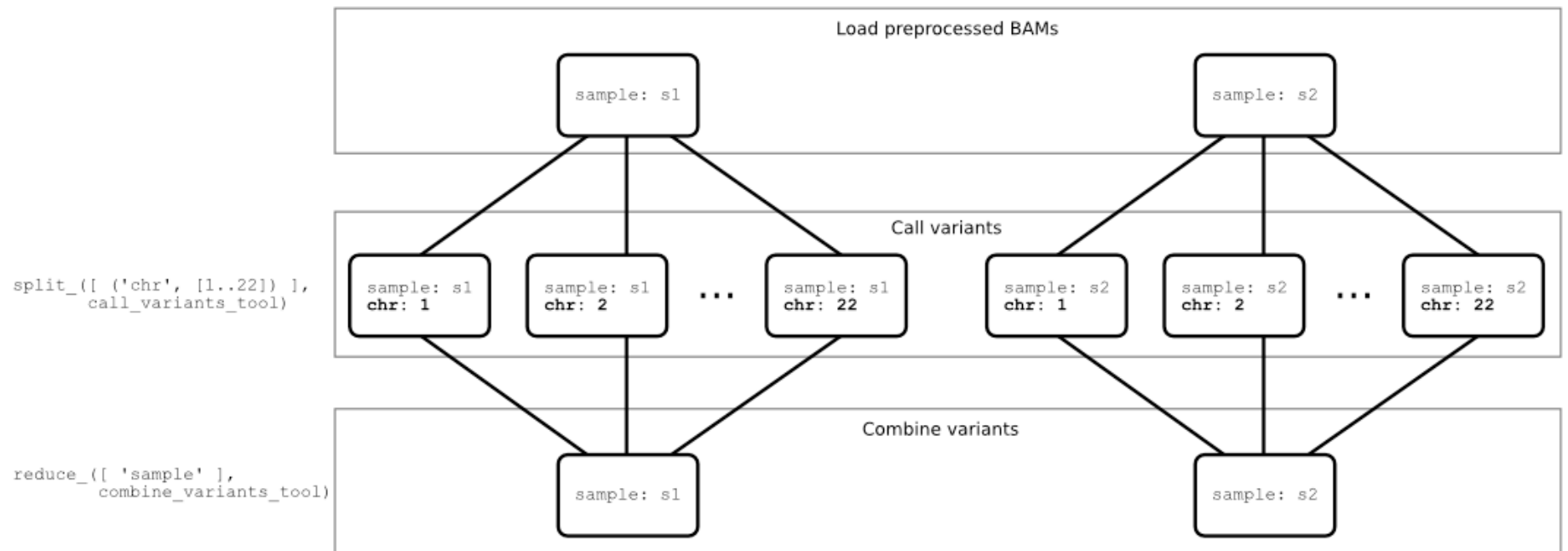


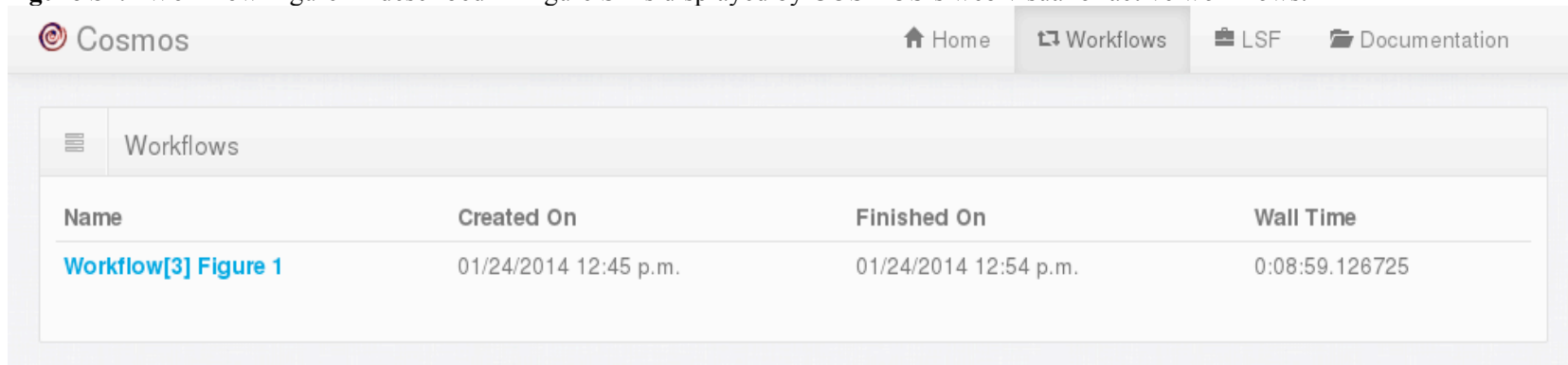
Figure S1. Example of tags used to parallelize variant calls by chromosome and variants agglomeration by sample

COSMOS introduces a “tag” system that associates a set of key-values (e.g., a sample ID, chunk ID, sequencer ID or other job parameter) to each task in the stage. These tags are used to associate a particular task for use and processing in COSMOS’ web interface, API and database. In this figure, the call variants stage of the workflow is parallelized by `split_` function with “chr” tag with values 1,2, ..., 22,(center row of Fig S1). The `split_` function inherits all parent tags (`sample`) and creates tasks for each distinct `chr` tag value. In the subsequent `reduce_` call (lower row of Fig S1), previous tasks are grouped by `sample` tag value thus only two tasks are created. Note that this construct allows `split_`, `reduce_` and other COSMOS tasks and jobs to be independent of ‘tagged’ objects such as number of “input files”.



COSMOS’ tag system improves upon Bpipe by allowing complicated workflow constructs such as the `split_` and `reduce_` DAG described above and visualized in Figure S1. In addition, Bpipe relies on a regular-expression with a single wild-card operator to split input files across parallel jobs. Consequently, in essence the information required to split jobs is encoded in file names. In contrast, COSMOS’s flexible tagging system uses the Python dictionary of key-value pairs on which to split and combine job outputs, even at distal stages in the workflow thus supporting a diverse collection of DAGs.

Figure S2. “Workflow Figure 1” described in Figure S1 is displayed by COSMOS’s web visual of active workflows.



The screenshot shows the COSMOS web interface. At the top left is the COSMOS logo. To the right are navigation links: Home, Workflows (highlighted), LSF, and Documentation. Below the navigation is a header for the 'Workflows' section. The main content is a table with the following data:

Name	Created On	Finished On	Wall Time
Workflow[3] Figure 1	01/24/2014 12:45 p.m.	01/24/2014 12:54 p.m.	0:08:59.126725

Figure S3. COSMOS Web visual of completed four stages in the workflow from Figure 1 along with runtime statistics. Note the number of tasks/job for each stage corresponds to the number of jobs in Figure 1(c).

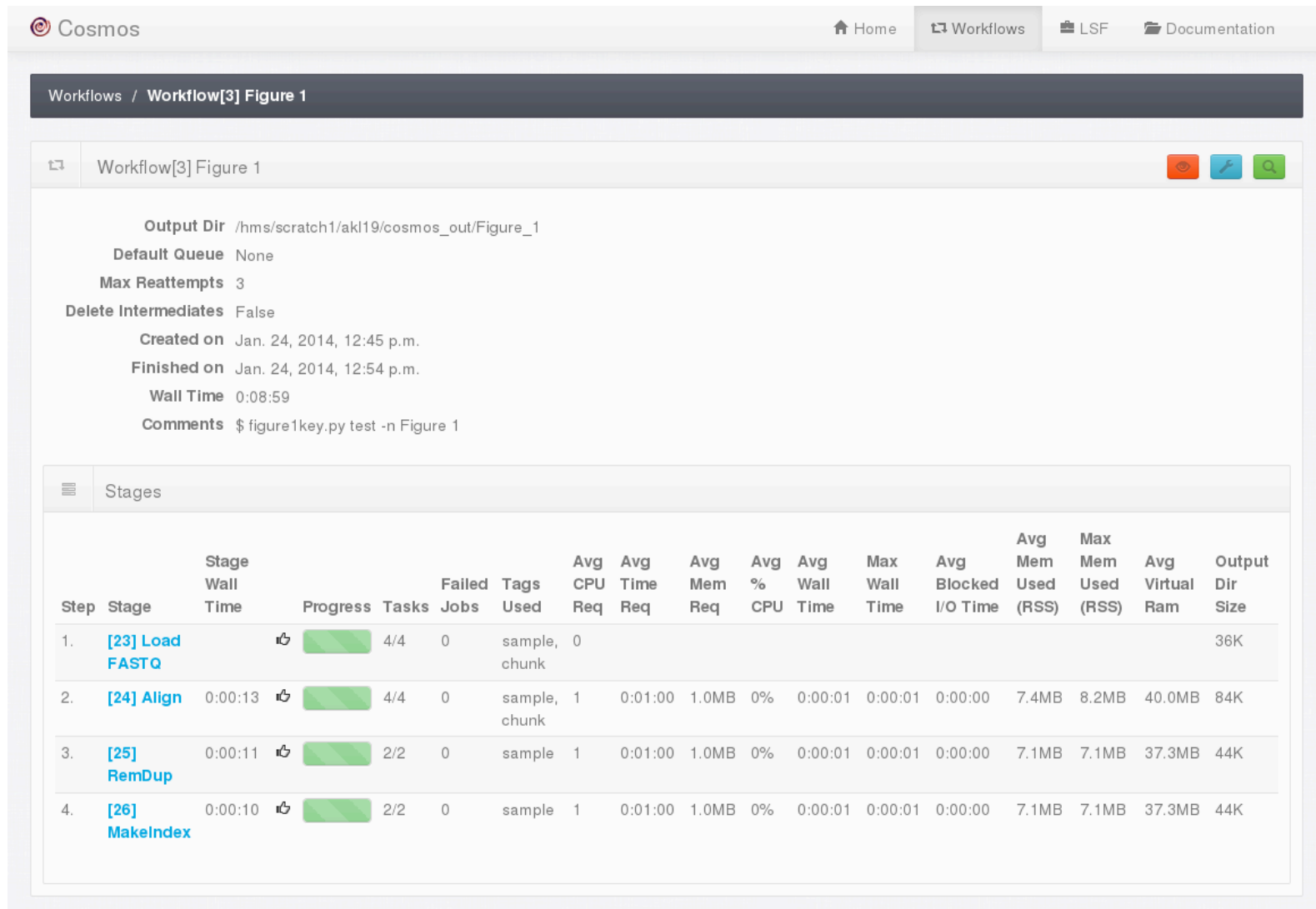


Figure S4. COSMOS Web visual showing a drilldown for the “Align” stage in the workflow. Each task/job sent via the DRMAA library to the underlying DRM (in this case, LSF) is displayed.

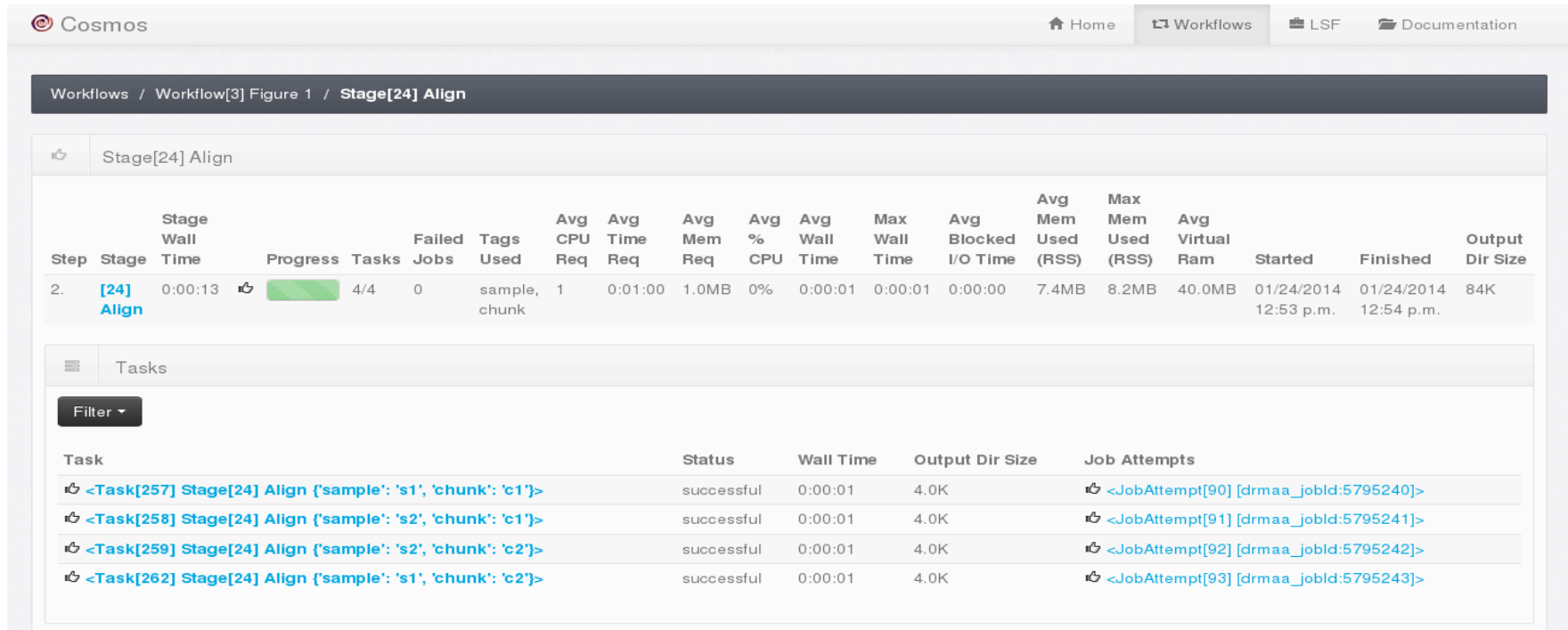


Figure S5. COSMOS Web visual of a ample in-progress COSMOS workflow of next-generation sequencing (NGS) analysis of three exomes, from loading of BAM files, through alignment, variant calling via the Genome Analysis Toolkit (GATK 3.x) to variant quality scoring of output of VCF files.

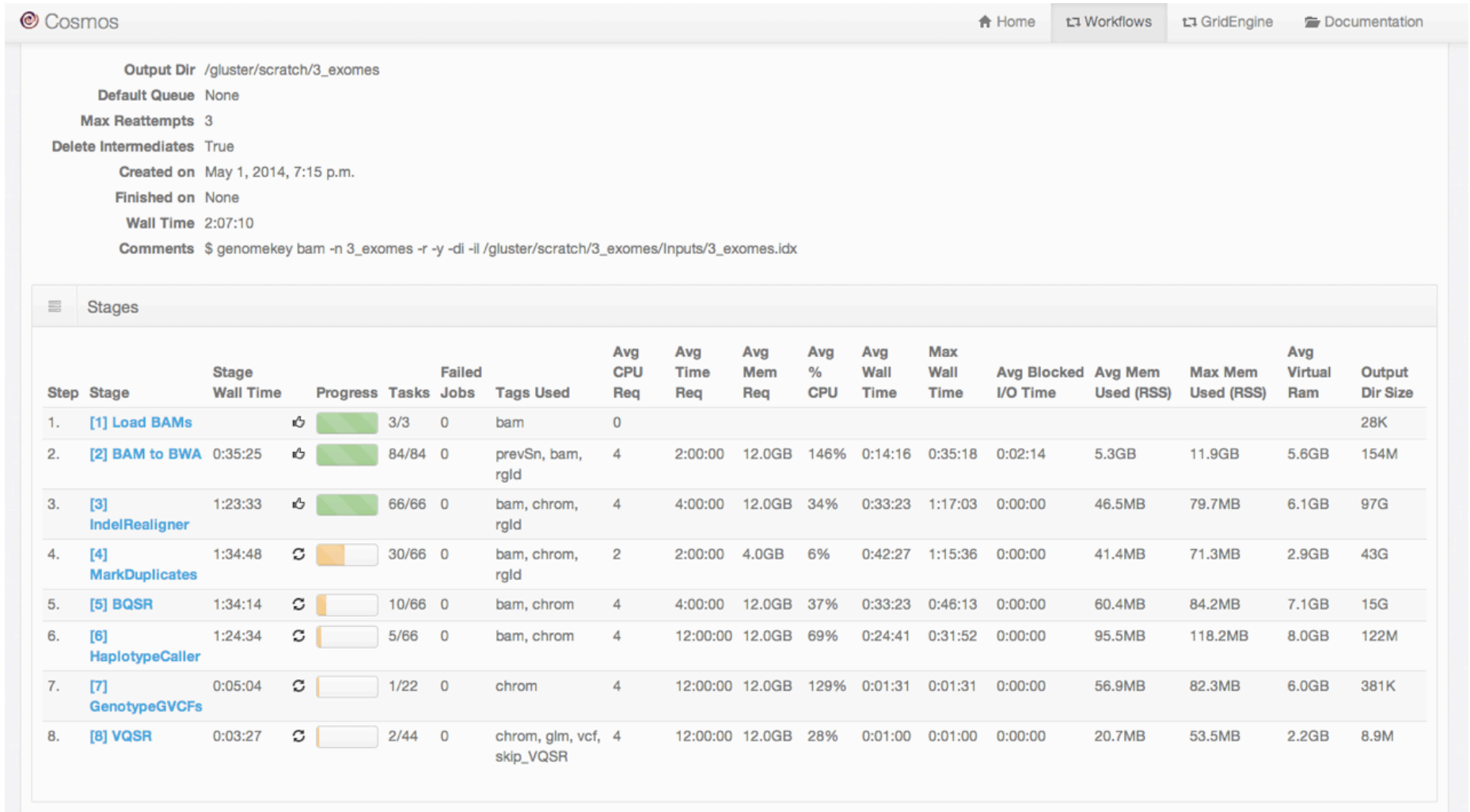


Figure S6. (A) Python code from the Pegasus tutorial (<http://pegasus.isi.edu/wms/docs/latest/tutorial.php#idp10013792>) implementing a “diamond” DAG with (B) roughly equivalent COSMOS code. Pegasus requires the programmer to manage the file names of inputs and outputs, while COSMOS handles this automatically.

```

A import sys, os

# Import the Python DAX library
os.sys.path.insert(0, "/usr/lib64/pegasus/python")
from Pegasus.DAX3 import *

# The name of the DAX file is the first argument
if len(sys.argv) != 2:
    sys.stderr.write("Usage: %s DAXFILE\n" % (sys.argv[0]))
    sys.exit(1)
daxfile = sys.argv[1]

# Create a abstract dag
print "Creating ADAG..."
diamond = ADAG("diamond")

# Add a preprocess job
print "Adding preprocess job..."
preprocess = Job(name="preprocess")
a = File("f.a")
b1 = File("f.b1")
b2 = File("f.b2")
preprocess.addArguments("-i",a,"-o",b1,"-o",b2)
preprocess.uses(a, link=Link.INPUT)
preprocess.uses(b1, link=Link.OUTPUT, transfer=False, register=False)
preprocess.uses(b2, link=Link.OUTPUT, transfer=False, register=False)
diamond.addJob(preprocess)

# Add left Findrange job
print "Adding left Findrange job..."
frl = Job(name="findrange")
c1 = File("f.c1")
frl.addArguments("-i",b1,"-o",c1)
frl.uses(b1, link=Link.INPUT)
frl.uses(c1, link=Link.OUTPUT, transfer=False, register=False)
diamond.addJob(frl)

# Add right Findrange job
print "Adding right Findrange job..."
frr = Job(name="findrange")
c2 = File("f.c2")
frr.addArguments("-i",b2,"-o",c2)
frr.uses(b2, link=Link.INPUT)
frr.uses(c2, link=Link.OUTPUT, transfer=False, register=False)
diamond.addJob(frr)

# Add Analyze job
print "Adding Analyze job..."
analyze = Job(name="analyze")
d = File("f.d")
analyze.addArguments("-i",c1,"-i",c2,"-o",d)
analyze.uses(c1, link=Link.INPUT)
analyze.uses(c2, link=Link.INPUT)
analyze.uses(d, link=Link.OUTPUT, transfer=True, register=False)
diamond.addJob(analyze)

# Add control-flow dependencies

```

```

B from cosmos.contrib.ezflow.dag import DAG, add_, sequence_, map_, apply_
from cosmos.contrib.ezflow.tool import INPUT

in_list = [ INPUT("f.a", tags={'sample': 1}) ]

dag = DAG().sequence_(
    add_(in_list),
    map_(preprocess),
    apply_(findrange_b1, findrange_b2, combine=True),
    map_(analyze)
)

```

```
print "Adding control flow dependencies..."
diamond.addDependency(Dependency(parent=preprocess, child=frr))
diamond.addDependency(Dependency(parent=preprocess, child=frr))
diamond.addDependency(Dependency(parent=frr, child=analyze))
diamond.addDependency(Dependency(parent=frr, child=analyze))
```

```
# Write the DAX to stdout
print "Writing %s" % daxfile
f = open(daxfile, "w")
diamond.writeXML(f)
f.close()
```

Table S1. Feature comparison between COSMOS and other workflow tools discussed in main text with respect to use in next-generation sequencing analysis (citations refer to references found in the main text)

	COSMOS	Bpipe (Sadedin et al. 2012)	Ruffus (Goodstadt, 2010)	Galaxy (Goecks et al., 2010)	Taverna (Wolstencroft et al. 2013)	Pegasus (Deelman et al., 2005)
Workflow language	Python	Groovy	Python	GUI	GUI	Java, Python, Perl
Workflow abstraction syntax	Yes	Yes	Yes	No	No	No ^[1]
DAG constructed at run-time	Yes	Yes	Yes	No	No	No
Support for complex parallelization	Yes	Yes	Yes	No	No	Yes
Realtime monitoring dashboard	Yes	No	No	Yes	Yes	Yes
SQL-like persistent database	Yes	No	No	Yes	No	No
Uses process exit codes ^[2]	Yes	No	No	Yes	Yes	Yes
Combine subworkflows	Yes	Yes	No	No	?	Yes
Support for >10k jobs ^[3]	Yes	No	No	No	No	Yes
Built-in DRM support	Yes	Yes	No	Yes	No	Yes
DRMAA ^[4] support	Yes	No	No	Yes	via plugin	No
Task DAG visualizer	Yes	No	No	Yes	Yes	Yes
Stage DAG visualizer	Yes	Yes	Yes	No	No	Yes
Topological execution ^[5]	Yes	No	No	Yes	Yes	Yes
Auto-delete intermediate files	Yes	No	No	No	No	No

^[1] Pegasus provides an API for constructing DAGs, but the workflow is not abstracted from the DAG itself; in effect, the Pegasus programmer must construct the DAG directly.

^[2] COSMOS' persistent database provides the ability to store task's exit codes, this solves some issues with Ruffus and Bpipe's reliance on output file timestamps is that failed jobs can be misinterpreted as successful when resuming a failed workflow.

^[3] Because Ruffus and Bpipe both create a thread per job, they may require more RAM than is available on modern machines for massive workflows. COSMOS was able to perform the GATK joint calling workflow with a dataset of 99 whole genomes.

^[4] Although Bpipe supports many popular DRMs, DRMAA has support for almost every widely used DRM including GridEngine, Condor, PBS/Torque, GridWay, PBS, and LSF.

^[5] COSMOS (and Galaxy and Taverna) execute a task as soon as its dependencies have successfully completed. Ruffus and Bpipe wait for an entire stage to complete before advancing.