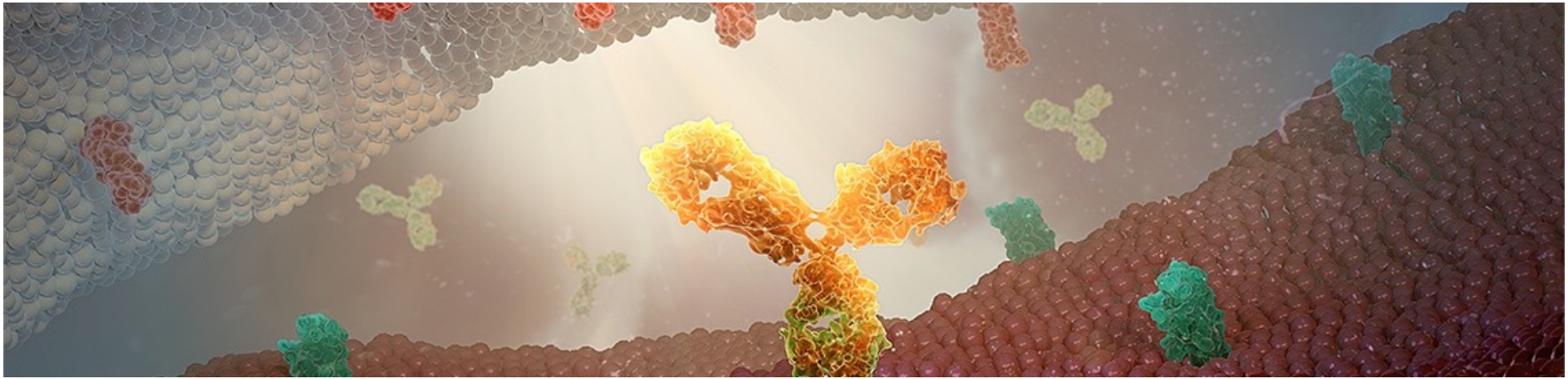
AstraZeneca

# bds Programming Language

## Scaling over 100,000 CPUs

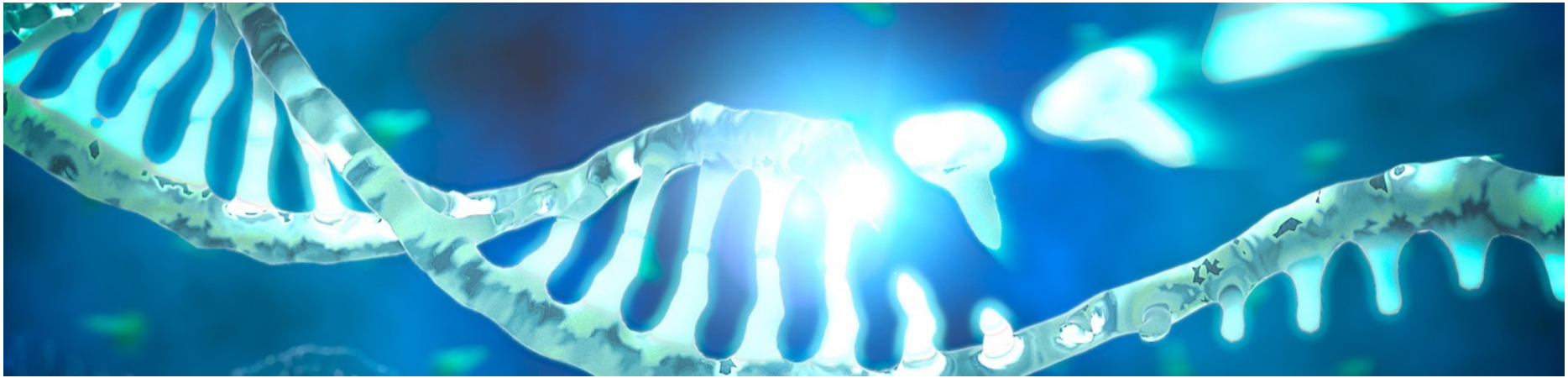
**Pablo Cingolani**
Director, NGS Informatics, Oncology

2021-09-22



# Motivation

- $A^3$: Algorithms, Analytics & AI
- Genomic & Imaging data processing
- Petabytes of data
- Different computing infrastructures: Cloud, Clusters/HPC, Servers, FPGA-accelerators
- Sequencing analyses involving thousands of CPUs are routine
- Computational Pathology analyses: Deep learning models on hundreds of GPUs



# bds

Domain specific language (DSL) for data analysis pipeline **orchestration**

## Design goals

- **Simple:** Easy to learn, solves task coordination
- **Architecture agnostic:** Program on a laptop, run on a cluster, or cloud
- **Robustness:** At large scales, errors manifest
- **Speedup development:** Fast prototyping, cleanup (process & files), reports, logging, command line parsing, strongly typed, testing, etc.

# bds: Simple

**Trivial to learn:** You should be able to start writing pipelines within minutes (the language is intended to be “boring”)

**Task:** Scaling and parallelism use simple models

**Task coordination** is solved at language level

# bds: Simple

**Trivial to learn:** You should be able to start writing pipelines within minutes (the language should look boring)

```
for(int i=0 ; i < 3 ; i++) {
  println “Hello world $i”
}
```

**Task:** Scaling and parallelism use simple models

**Task coordination** is solved at language level

# bds: Simple

Trivial to learn: You should be able to start writing pipelines within minutes (the language should look boring)

**Task:** Scaling and parallelism use simple models

Task **coordination** is solved at language level

```
for(int i=0 ; i < 3 ; i++) {
  task echo “Process $i”
}
wait    # Wait for all tasks finish
println “Done”
```

```
$ bds z.bds
Process 2
Process 0
Process 1
Done
$
```

# bds: Simple

Trivial to learn: You should be able to start writing pipelines within minutes (the language should look boring)

Task: Scaling and parallelism use simple models

**Task coordination** is solved at language level

```
# Align reads for each sample
string[] bams
for(int i=0 ; i < numberSamples ; i++) {
  fastq := “fastq_” + i + “.fq”
  bam := “aligned_” + i + “.bam”
  task(bam <- fastq) {
    sys bwa mem hg38 $fastq \
          | samtools sort > $bam
  }
  bams += bam
}
# Joint genotyping
bamsStr := bams.join(‘ ‘)
vcf := “variants.vcf”
task(vcf <- bams) {
  sys joint_genotype $ bamsStr > $vcf
}
```

# bds: Architecture agnostic

A bds program runs on different systems without modification.

bds is abstracts underlying infrastructure

- Single server / laptop
- Cluster: SGE, MOAB, Slurm, Torque, etc.
- Cloud
- Server farm

bds abstracts execution environments

# bds: Architecture agnostic

A bds program runs on different systems without modification.

bds is abstracts underlying infrastructure

- Single server / laptop
- Cluster: SGE, MOAB, Slurm, Torque, etc.
- Cloud
- Server farm

bds abstracts execution environments

```
for(int i=0 ; i < 3 ; i++) {
  task echo “Process $i”
}
wait    # Wait for all tasks finish
println “Done”
```

```
# Execute on my local laptop
$ bds z.bds
Process 0
Process 2
Process 1
Done
$
```

# bds: Architecture agnostic

A bds program runs on different systems without modification.

bds is abstracts underlying infrastructure

- Single server / laptop
- Cluster: SGE, MOAB, Slurm, Torque, etc.
- Cloud
- Server farm

bds abstracts execution environments

```
# Execute on an SGE cluster
$ bds -s sge z.bds
Process 0
Process 2
Process 1
Done
$
```

```
# Execute on a SLURM cluster
$ bds -s slurm z.bds
Process 0
Process 1
Process 2
Done
$
```

# bds: Architecture agnostic

A bds program runs on different systems without modification.

bds is abstracts underlying infrastructure

- Single server / laptop
- Cluster: SGE, MOAB, Slurm, Torque, etc.
- Cloud
- Server farm

bds abstracts execution environments

```
# Execute on an SGE cluster
$ bds -s sge z.bds
Process 0
Process 2
Process 1
Done
$
```

```
# Execute on a SLURM cluster
$ bds -s slurm z.bds
Process 0
Process 1
Process 2
Done
$
```

Note: bds invokes the appropriate **qsub** or **sbatch** commands, monitors jobs in cluster, collect post-mortem information if the job fails

# bds: Architecture agnostic

A bds program runs on different systems without modification.

bds is abstracts underlying infrastructure

- Single server / laptop
- Cluster: SGE, MOAB, Slurm, Torque, etc.
- Cloud
- Server farm

bds abstracts execution environments

```
# Execute on an SGE cluster
$ bds -s aws z.bds
Process 0
Process 2
Process 1
Done
$
```

# bds: Architecture agnostic

A bds program runs on different systems without modification.

bds is abstracts underlying infrastructure

- Single server / laptop
- Cluster: SGE, MOAB, Slurm, Torque, etc.
- Cloud
- Server farm

bds abstracts execution environments

```
# Execute on an SGE cluster
$ bds -s aws z.bds
Process 0
Process 2
Process 1
Done
$
```

Note: bds will…

- Create an instance
  - Create a start-up script that executes the task
  - Create a trap in startup script to shut-down on error
  - Set the instance as "Terminate on shutdown"
  - Retries if AWS run instance fails
- Create an SQS queue
- The instance starts up
  - Executes the task
  - Redirect STDOUT / STDERR to SQS queue
  - Send task's EXIT code to SQS (when task finishes)
- Local bds process
  - Monitors SQS, shows STDOUT / STDERR messages
  - Marks task as finished when EXIT code is received

# bds: Architecture agnostic

A bds program runs on different systems without modification.

bds is abstracts underlying infrastructure

- Single server / laptop
- Cluster: SGE, MOAB, Slurm, Torque, etc.
- Cloud
- Server farm

bds abstracts execution environments

```
# Execute on a cluster
task(out1 <- in1, system := ‘slurm’) {
  sys some_cmd $in1 > $out1
}

# Execute on a Cloud
task(out2 <- in2, system := ‘aws’) {
  sys another_cmd $in2 > $out2
}

# Execute as a local process
task(out <- [out1, out2]) {
  sys join_outs $out1 $out2 > $out
}
```

# bds: Architecture agnostic

A bds program runs on different systems without modification.

bds is abstracts underlying infrastructure

- Single server / laptop
- Cluster: SGE, MOAB, Slurm, Torque, etc.
- Cloud
- Server farm

bds abstracts execution environments

Note: bds coordinates across multiple systems (Slurm, AWS and local)

```
# Execute on a cluster
task(out1 <- in1, system := ‘slurm’) {
  sys some_cmd $in1 > $out1
}

# Execute on a Cloud
task(out2 <- in2, system := ‘aws’) {
  sys another_cmd $in2 > $out2
}

# Execute as a local process
task(out <- [out1, out2]) {
  sys join_outs $out1 $out2 > $out
}
```

# bds: Architecture agnostic

A bds program runs on different systems without modification.

bds is abstracts underlying infrastructure

- Single server / laptop
- Cluster: SGE, MOAB, Slurm, Torque, etc.
- Cloud
- Server farm

bds abstracts execution environments

```
# Absolute serialization
# Continue execution in AWS
task(out <- in, system := ‘aws’) {
  println “This is bds code”
  myFunction(in, out)
}
```

Note: bds will…

- **Serialize** the current VM state
- Transfer it to the instance
- **Continue** executing bds code where it left (inside the task statement)

# bds: Robust

Lazy processing: Do not re-run tasks if not needed

Automatic re-runs failed tasks

Absolute serialization (‘checkpoint’)

Check program exit code
Check output files exists (and non-empty)
Check jobs that disappeared from clusters
Check Cloud instances are alive
Retry creating Cloud instances

# bds: Robust

**Lazy processing**: Do not re-run tasks if not needed

Automatic re-runs failed tasks

Absolute serialization (‘checkpoint’)

Check program exit code
Check output files exists (and non-empty)
Check jobs that disappeared from clusters
Check Cloud instances are alive
Retry creating Cloud instances

```
# Lazy processing example
in := “in.txt”
out := “out.txt”
task(out <-in) {
 sys echo “Processing $in”
 sys cat $in > $out
}
wait
println “Done”
```

```
# First time we run task is executed
$ bds z.bds
Processing in.txt
Done
```

```
# Run again, task is not executed
$ bds z.bds
Done
```

# bds: Robust

Lazy processing: Do not re-run tasks if not needed

Automatic re-runs failed tasks

Absolute serialization (‘checkpoint’)

Check program exit code
Check output files exists (and non-empty)
Check jobs that disappeared from clusters
Check Cloud instances are alive
Retry creating Cloud instances

```
# Checkpoint example
for(int i=0 ; i < 10 ; i++) {
    if(i == 5) {
        println "Checkpoint"
        checkpoint "my.chp"
    }
    println "Counting $i"
}
```

```
# Restart execution from checkpoint file
$ bds -r my.chp
Counting 5
Counting 6
Counting 7
Counting 8
Counting 9
```

# bds: Robust

Lazy processing: Do not re-run tasks if not needed

Automatic re-runs failed tasks

Absolute serialization (‘checkpoint’)

Check program exit code
Check output files exists (and non-empty)
Check jobs that disappeared from clusters
Check Cloud instances are alive
Retry creating Cloud instances

```
in := ’in.txt’
out := ‘out.txt’
task(out <- in) sys touch $out
```

```
$ bds z.bds
. . .
ERROR: Task failed:
. . .
Output file checks : 'Error: Output file
'out.txt' has zero length.'
```

# bds: Speedup development

Cleanup: Processes, files, instances
Summary reports
Command line parsing & help
Logging
Remote files
Strongly typed
Multithreading: par
Built-in debugger
Built-in unit testing
Dry run
Declarative style: goal
Task resources
Detached tasks

# bds: Speedup development

Cleanup: Processes, files, instances
Summary reports
Command line parsing & help
Logging
Remote files
Strongly typed
Multithreading: par
Built-in debugger
Built-in unit testing
Dry run
Declarative style: goal
Task resources
Detached tasks

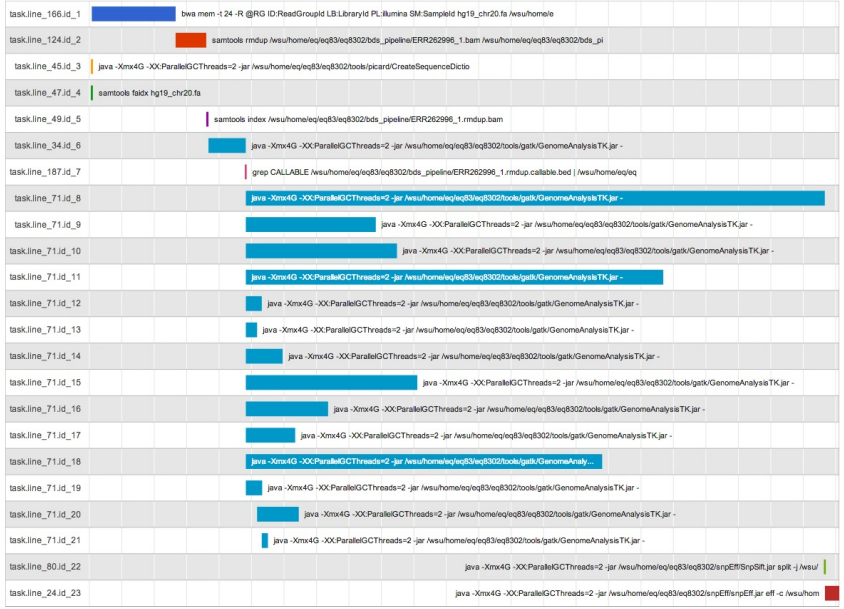
```
in := 'in.txt'
out := 'out.txt'
task(out <- in) sys long_cmd $out
```

```
$ bds z.bds
. . .
# Press Ctrl-C will
```

Note: When you press Ctr-C bds will:
- Delete output file/s
- Kill any processes running a tasks
- Kill any cluster jobs running tasks
- Terminate any instances running tasks
- Delete cloud queues

# bds: Speedup development

Cleanup: Processes, files, instances
Summary reports
Command line parsing & help
Logging
Remote files
Strongly typed
Multithreading: par
Built-in debugger
Built-in unit testing
Dry run
Declarative style: goal
Task resources
Detached tasks



# bds: Speedup development

Cleanup: Processes, files, instances
Summary reports
Command line parsing & help
Logging
Remote files
Strongly typed
Multithreading: par
Built-in debugger
Built-in unit testing
Dry run
Declarative style: goal
Task resources
Detached tasks

```
in := ’in.txt’ help Input file name
println “Input is: $in”
```

```
$ bds z.bds –h
Command line options 'z.bds' :
     -in <string>  : Input file name
```

```
$ bds z.bds -in my_file
Input is: my_file
```

# bds: Speedup development

Cleanup: Processes, files, instances
Summary reports
Command line parsing & help
Logging
Remote files
Strongly typed
Multithreading: par
Built-in debugger
Built-in unit testing
Dry run
Declarative style: goal
Task resources
Detached tasks

```
in := 's3://my_bucket/in.txt'
out := 'out.txt'
task(out <- in) sys cat $in > $out
```

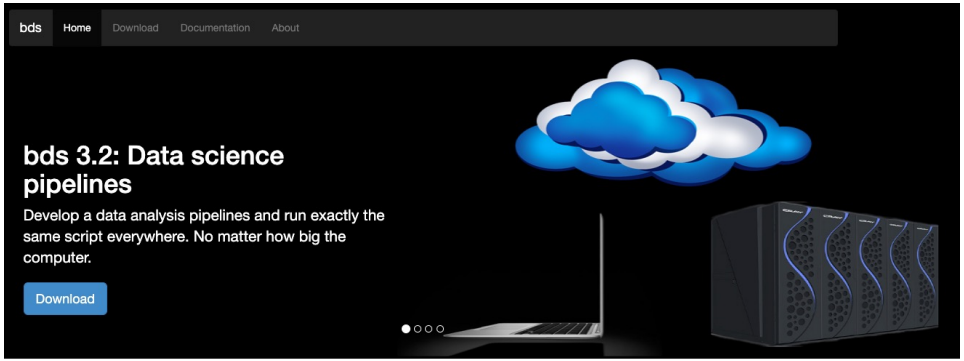
```
$ bds z.bds
```

Note: bds will
- Check the remote file in S3
- Compare timestamps against 'out.txt'
- Download the S3 file
- Change 'cat' command (downloaded file)
- Execute the task

# Conclusions

- bds is a Domain Specific Language (DSL) for data analysis pipeline **orchestration**
- bds is mature and well tested
- bds allows to scale data analysis pipelines to thousands of CPUs & GPUs
- Design goals: Simple, Architecture agnostic, Robustness, Speedup development

# bds: Web, Docs, GitHub, Paper

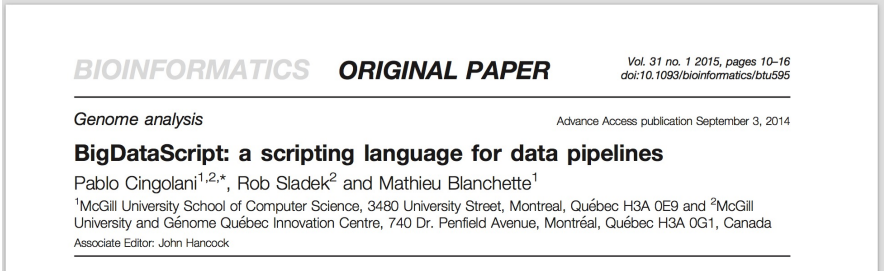



Data pipelines

Develop

Run

https://github.com/pcingola/bds



BIOINFORMATICS **ORIGINAL PAPER**

*Vol. 31 no. 1 2015, pages 10–16*
*doi:10.1093/bioinformatics/btu595*

*Genome analysis*

Advance Access publication September 3, 2014

**BigDataScript: a scripting language for data pipelines**

Pablo Cingolani[1,2,*], Rob Sladek[2] and Mathieu Blanchette[1]

[1]McGill University School of Computer Science, 3480 University Street, Montreal, Québec H3A 0E9 and [2]McGill University and Génome Québec Innovation Centre, 740 Dr. Penfield Avenue, Montréal, Québec H3A 0G1, Canada

Associate Editor: John Hancock

# Thank you!