eHive Workshop

part 2. How to create pipelines

Leo Gordon

Before we begin, please:

- Make sure you have correct setup from part 1
- Get a username/password on the training MySQL server
- Update ensembl-hive repository to pick fresh bug fixes

```
export EHIVE_HOST=mysql-ehive-training.ebi.ac.uk
export EHIVE_PORT=4427
export EHIVE_USER=trainXX
export EHIVE_PASS=pswXX
```

- Breaks?

Nested loops and dimensions

```
foreach my $species (@all_species) {
    foreach my $chromosome (@{ $species->get_Chromosomes }) {
        foreach my $gene (@{ $chromosome->get_Genes }) {
            foreach my $transcript (@{ $gene->get_Transcripts }) {
                foreach my $exon (@{ $transcript->get_Exons }) {
                    # is this the Exon I was really looking for?
                }
            }
        }
    }
}
```

- A regular program has to fold all of this 5D into 1T
- A well-designed pipeline can “turn” some of these dimensions into “space” of the farm
- When planning a pipeline, first draw a diagram and think about these dimensions and their real dependence.

Modularity of pipelines. PipeConfigs & Runnables

- A Hive pipeline is defined in a PipeConfig file which references one or more Runnable modules.

- Many tasks can be solved by using “universal” Runnables provided by the Hive (SystemCmd, SqlCmd, JobFactory, Dummy), but sometimes you have to write your own application-specific Runnables.

- We shall learn to use universal Runnables before making our own
  (however it may be the opposite of what you do in practice)

- Hive’s “universal” Runnables live here:

  $ENSEMBL_CVS_ROOT_DIR/ensembl-hive/modules/Bio/EnsEMBL/Hive/RunnableDB/

  They are written in Perl.

- Hive’s PipeConfig files live here:

  $ENSEMBL_CVS_ROOT_DIR/ensembl-hive/modules/Bio/EnsEMBL/Hive/PipeConfig/

  They are written in a subset of Perl.

The simplest pipeline : AnyCommands_conf.pm

- Please open the file - this is the smallest PipeConfig possible:

```
use base ('Bio::EnsEMBL::Hive::PipeConfig::HiveGeneric_conf');#  or subclass

sub pipeline_analyses {
    return [
        {   -logic_name    => 'perform_cmd',
            -module        => 'Bio::EnsEMBL::Hive::RunnableDB::SystemCmd',
        },
    ];
}
```

```
$ init_pipeline.pl AnyCommands_conf.pm
```

```
$ generate_graph.pl -url $EHIVE_URL -out any_empty.png
```

or use http://guihive.ebi.ac.uk:8080/ instead

AnyCommands_conf.pm (2)

- No jobs - we will have to seed them:

```
$ seed_pipeline.pl -url $EHIVE_URL -logic_name perform_cmd
\
    -input_id '{"cmd" => "echo Hello, world"}'
```

- and run:

```
$ runWorker.pl -url $EHIVE_URL
```

- Practical to a certain extent (analysis_capacity, batch_size, resources)

Analysis-wide parameters and substitution

- We can define values for old parameters and create new ones:

```
sub pipeline_analyses {
    return [
        {   -logic_name    => 'perform_cmd',
            -module        => 'Bio::EnsEMBL::Hive::RunnableDB::SystemCmd',
            -parameters => {
                 "cmd" => "gzip #filename# ",
            },
        },
    ];
}
```

- Exercise: seed a few jobs and run them (you can copy some compressible files from ~lg4/work/pdfs)

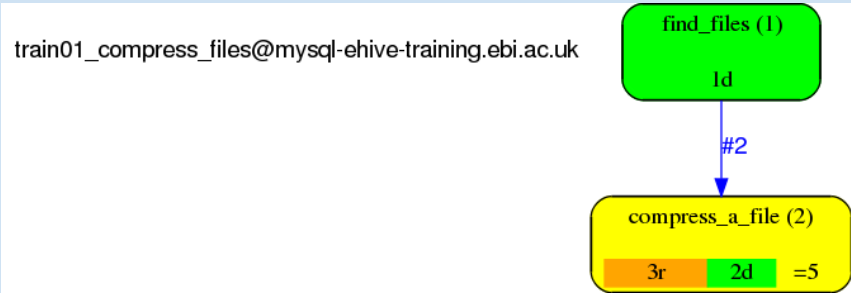
- Automated seeding of jobs?

```
for filename in `find pdfs/ -name '*.pdf'` ; do seed_pipeline.pl -url $EHIVE_URL \
     -logic_name perform_cmd -input_id "{ 'filename' => '$filename' }";
done
```

Factories and dataflow : CompressFiles_conf.pm

- Dataflow Rules can be used to make Jobs seed other Jobs
- Factory is an Analysis whose only aim is to seed other Jobs, create a “fan”, turn time into space
- Higher level input

```
sub pipeline_analyses {
    return [
        {   -logic_name => 'find_files',
            -module     => 'Bio::EnsEMBL::Hive::RunnableDB::JobFactory',
            -parameters => {
                'inputcmd'     => 'find #directory# -type f ',
                'column_names' => [ 'filename' ],
            },
            -flow_into => {
                2 => [ 'compress_a_file' ],
            },
        },
        {   -logic_name    => 'compress_a_file',
            -module        => 'Bio::EnsEMBL::Hive::RunnableDB::SystemCmd',
            -parameters    => {
                'cmd'       => 'gzip #filename#',
            },
            -analysis_capacity => 4,
        },
    ];
}
```



Dataflow conventions

- Each Dataflow Event is a pair (branch_number, hash_of_parameters+).

- Each Runnable has its own set of branch_numbers
  that it may emit Dataflow Events into.

- “Reserved” branch numbers that have their own meaning:
  - 1 is almost always present, it is the “continuation” after Job is ‘DONE’
  - 2 is used by many Factory Runnables to emit the “fan” of Jobs
  - -1 : “postmortem dataflow after MEMLIMIT” on LSF
  - -2 : “postmortem dataflow after RUNLIMIT” on LSF

- Check the Runnable (documentation or code) to make sure
  the dataflow you are wiring is live. What happens if it’s not?

- “Black boxing” exercise: modify **`MemlimitTest_conf.pm`** to dataflow into another Analysis with more memory if not enough.

pipeline_wide_parameters()
and the order of precedence of parameters

- The source of parameters is unknown to Jobs

```
sub pipeline_wide_parameters {
    my ($self) = @_;
    return {
        %{$self->SUPER::pipeline_wide_parameters},

        'gzip_flags'    => '',
        'directory'     => '.',
        'only_files'    => '*',
    };
}
```

- Parameters can be:
  - “local” to the Job (belonging/sent to the Job itself or its “stack” of ancestors)
  - analysis-wide
  - pipeline-wide
  - defaults set in the Runnable’s code

- Exercise: abstract the compressor (‘compress’, ‘gzip’, ‘bzip2’, ‘xz’ etc) out

Templates: the other kind of glue

- Runnables have fixed parameter names for input and output - comparison with Perl subroutine calls:
  - \+ more flexible - you can specify certain parameters and not others
  - \+ less error-prone - if you add parameters, there is no need to reshuffle them
  - − you need “glue” to link analyses together

- Two kinds of glue:
  - input transformation using parameter substitution:

```
'cmd' => 'gzip #filename#'
```

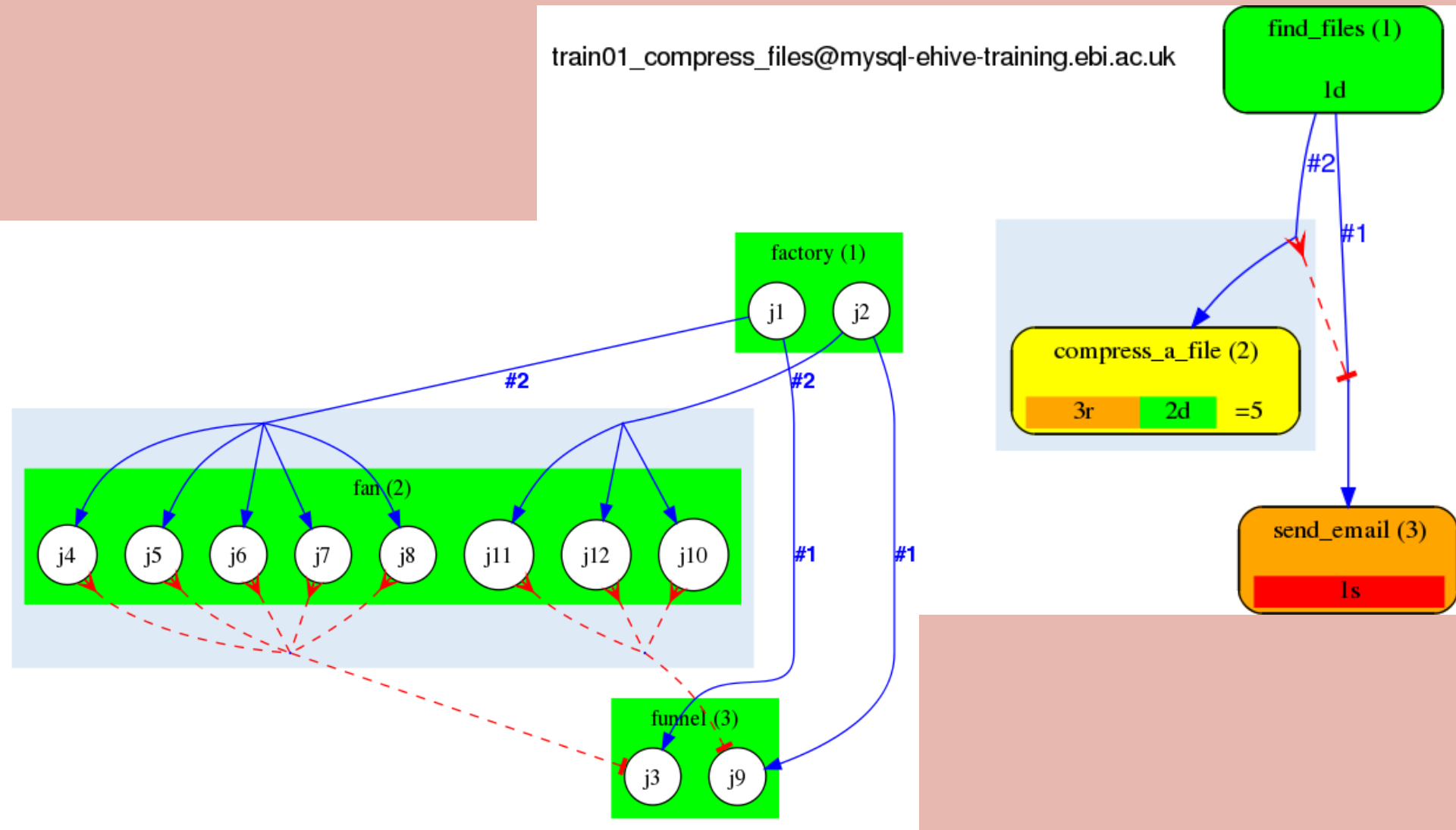
  - output transformation using templates:

```
2 => { 'compress_a_file' => {
            'input_filename' => '#output_filename#', # rename
            'check_input_once' => 1,                 # specific mode
            'gzip_flags' => '#gzip_flags#',          # propagation
          },
         'another_analysis' => undef,   # no template - use as is
},
```

- Templates work the same way independently of Dataflow’s *destination type*

Regaining single thread of control: semaphored dataflow

- Built-in mechanism for converging individual threads back together.
- Based on semaphores that can block an individual job by a set of prerequisite jobs.



Semaphored dataflow in a PipeConfig

- Creating a funnel Analysis : let the pipeline send us a notification:

```
{   -logic_name    => 'report',
      -module        => 'Bio::EnsEMBL::Hive::RunnableDB::NotifyByEmail',
      -meadow_type   => 'LOCAL',  # NB: farm nodes may not support sendmail
      -parameters    => {
          'email'   => $ENV{'USER'} . '@ebi.ac.uk',
          'subject' => 'pipeline has finished',
          'text'    => 'done compressing files in #directory#',
      },
},
```

- Linking two rules together happens in the emitting Analysis:

```
-flow_into => {
    '2->A' => { 'compress_a_file' =>
                 { 'filename' => '#filename#', 'gzip_flags' => '#gzip_flags#' },
         },
    'A->1' => [ 'send_email' ],
}
```

- Try running?

Capturing data from a command : JobFactory

- Both SystemCmd and SqlCmd only run your command, no output is captured in any structured way.
  - So SqlCmd is usually used to INSERT, DELETE, UPDATE, CREATE/ALTER/DROP TABLE, but not with SELECT.

- JobFactory is not specifically creating Jobs - it simply transforms streams of “things” into Dataflow Events that may be converted into Jobs, stored in database tables, or accumulated. It is the wiring that defines what happens next.

```
{   -logic_name    => 'post_compress_size',
    -module        => 'Bio::EnsEMBL::Hive::RunnableDB::JobFactory',
    -parameters    => {
        'inputcmd'      => "wc -c #filename#.gz | sed -e 's/^ *//' ",
        'delimiter'     => ' ',
        'column_names'  => [ 'comp_size', 'comp_filename' ],
    },
},
```

- Insert it correctly. Have we captured anything? We have. How to pass it outside?

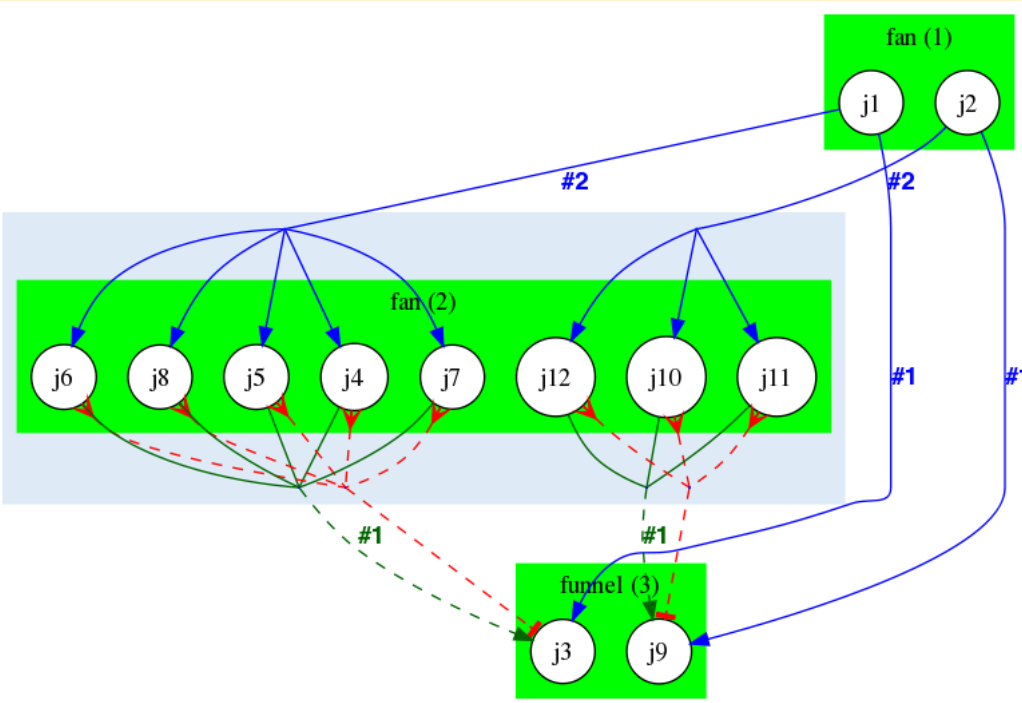
Accumulating data from a semaphore group

- How do we pass the data from the box into the funnel?
- The data can be passed from any job within the box into the correct funnel Job
- Different structures or combinations can be accumulated (hashes, arrays, piles, multisets)
- pseudo-Analysis names as targets for Dataflow (with or without templates).

```
':////accu?hash_name={key_name}'

':////accu?array_name=[index_name]'

':////accu?pile_name=[]'

':////accu?multiset_name={}'
```

- see LongMult_conf for example.

Flow the data into accu (which branch?)



Advanced parameter substitution : expressions

- What if we want to compute a value of #alpha#+1 rather than just a string?

```
'alpha_plus_one' => '#expr($alpha+1)expr#'
```

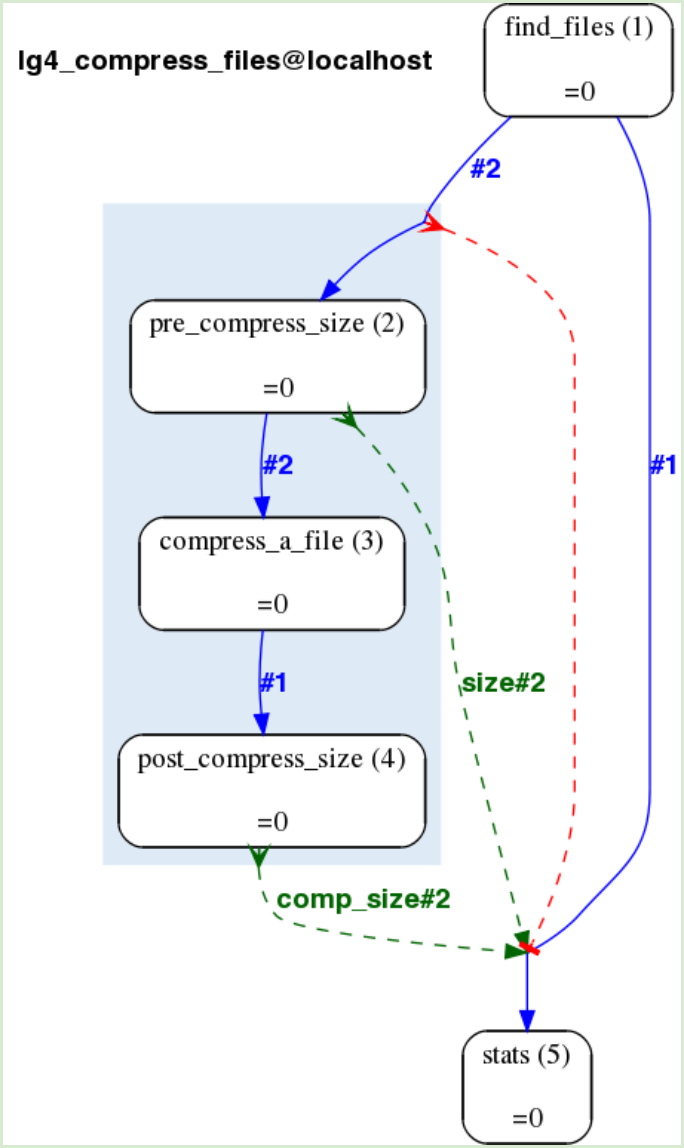
- Any Perl expression can be evaluated as follows:
  - first, $alpha will be text-substituted with the value of alpha parameter
  - then the resulting string will be evaluated
  - so put a space between dollar and the name ($ beta) if you want standard Perl interpretation of the variable

- We can flatten accumulated structures (that are not scalars) into scalars using #expr()expr# . For example,

```
'min_comp_size' => '#expr(min values %{$comp_size})expr#',
'max_comp_size' => '#expr(max values %{$comp_size})expr#',
'text'    => 'compressed sizes between #min_comp_size# and #max_comp_size#',
```

Exercise: accumulation + substitution

- Let's put it all together:
  - Factory on a directory to dataflow single filenames
  - compute their sizes and accumulate them
  - compress the files
  - compute the compressed sizes and accumulate
  - funnel flattens the accumulated structures and emails you the report



- Have a look at my PipeConfig/CompressFilesWithReport_conf.pm

Manual parameter propagation vs parameter stack

- There used to be only one way of propagating parameters into sub-branches - using templates. It gave full control over which parameters a Job sees, but was non-intuitive in comparison to structured programs with scoped variables.

- Open both LongMult_conf and LongMultSt_conf . Notice the difference.

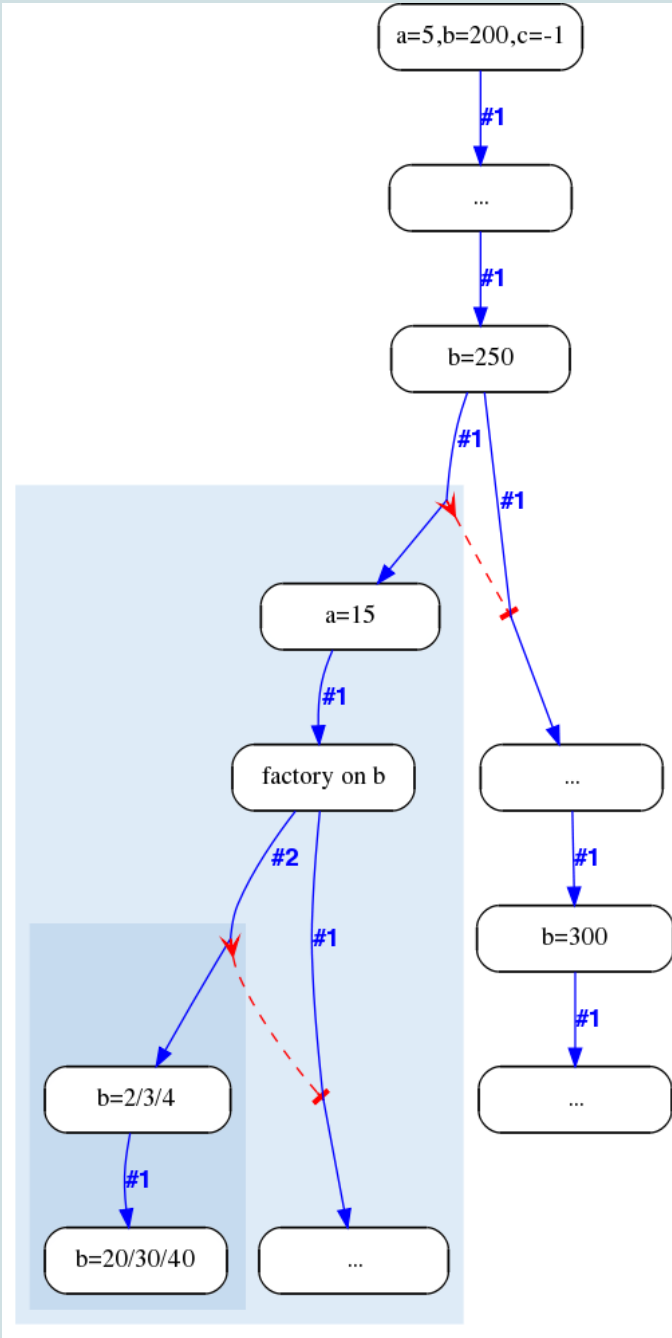
- Parameter stack approach may be less efficient but much more intuitive and easier to use for programmers. Once a variable was dataflown down a thread, its value becomes available to all its descendants until the end of that dataflow thread (Hive equivalent of end-of-scope).

- Parameter stack support is off by default, can be switched on in PipeConfig:

```
sub hive_meta_table {
    my ($self) = @_;
    return {
        %{$self->SUPER::hive_meta_table},
        'hive_use_param_stack'  => 1,
    };
}
```

How parameter stack works

```
{
    a=5 ; b=200 ; c=-1;
    …
    b=250 ;
    …
        {
            a=15;
            for b in (2,3,4) {
                b *= 10;
            }
            …
        }
    …
    b=300;
}
```



Writing your own Runnable. Lifecycle of a Job

- A Job goes through the following states:
  - [SEMAPHORED] -- if they are created in pre-blocked state
  - READY -- can be claimed by Workers
  - CLAIMED -- for a short period to ensure no race condition with other Workers
  - [PRE_CLEANUP] -- method -- mostly file/db cleanup after prev. attempt
  - FETCH_INPUT -- method -- checking parameters and database activity
  - RUN -- method -- main functionality, ideally mute
  - WRITE_OUTPUT -- method -- mostly writing into databases, dataflow
  - [POST_CLEANUP] -- method -- mostly memory cleanup
  - DONE -- this is how they all should be
  - [FAILED] -- if exhausted all attempts
  - [PASSED_ON] -- if garbage-collected from a killed Worker

- A Runnable can redefine some of the following “virtual” methods:

  - param_defaults() -- a hash of the lowest level defaults in parameter precedence
  - pre_cleanup()
  - fetch_input()
  - run()
  - write_output()
  - post_cleanup()

Writing your own Runnable. Available API

- To get access to all the API available to the Runnable, it has to inherit from Process, directly or indirectly:

```
use base ('Bio::EnsEMBL::Hive::Process');      # or your group's subclass
```

- The main cohesive material of the Hive system is the API that deals with parameter retrieval, storage and propagation. It is closely linked with dataflow mechanism.

  - my $alpha = $self->param('alpha'); # getter
  - my $alpha = $self->param_required('alpha'); # strict getter
  - $self->param('beta', $beta); # setter
  - if ($self->param_is_defined('gamma')) { … } # check
  - $self->dataflow_output_id({ 'alpha' => 1.5, 'gamma' => 5 }, 3); # send event
  - $self->warning('I got a strange feeling I am in an infinite loop…'); # not warn
  - die 'all gone wrong'; # record this message in log_message
  - $self->throw(); # or any other die-based reporter
  - $self->input_job->incomplete(0 | 1);
  - $self->input_job->transient_error(0 | 1);
  - $self->input_job->lethal_for_worker(0 | 1);

Writing your own Runnable. Exercise

- Please study the LongMult example Runnables, they are always up to date with the development

- Exercise: write a Runnable that takes in an array of LONG numbers and adds them all together. It may need its own “driver” PipeConfig.

- Prove that (A1+A2+A3+...+An)*B == A1*B+A2*B+A3*B+...+An*B by adding your new Runnable to LongMultSt_conf.

Questions?

Acknowledgements

Matthieu Muffato and Miguel Pignatelli

Current and previous members of Compara team

All users of eHive system for testing, feedback and ideas

Paul Flicek, Steve Searle and the entire Ensembl Team

Funding

wellcome trust

EMBL

National Human Genome Research Institute

BBSRC bioscience for the future

European Commission Framework Programme 7

SEVENTH FRAMEWORK PROGRAMME

Quantomics

From Sequence to Consequence : Tools for the Exploitation of Livestock Genomes

GEN2PHEN

BLUEPRINT epigenome