eHive Workshop

part 3: writing your own Runnables

Leo Gordon

EMBL-EBI

wellcome trust sanger institute

States of a Job

- A Job is a parametrised Storable instance of a Runnable.
  It is fully represented in the Hive database (job table, job_id, foreign keys, etc)

- 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

Lifecycle of a Runnable/Job

- Hive Runnables inherit from `'Bio::EnsEMBL::Hive::Process'` (or its descendents). It gives them two things:

  - they get access to Hive API (the visible part of which is parameter management)
  - they acquire a lifecycle() subroutine that calls the following “virtual” methods:

    - param_defaults()   # a hash of the lowest level defaults in parameter precedence
    - pre_cleanup()   # is only called for retry_counts>0, mainly to clean up files
    - fetch_input()
    - run()
    - write_output()
    - post_cleanup()   # mainly to clean up memory after all values of retry_count

- standaloneJob.pl is a script to run a parametrised Runnable without a database at all:

```
standaloneJob.pl Bio::EnsEMBL::Hive::RunnableDB::SystemCmd -cmd 'ls -l'

standaloneJob.pl Bio::EnsEMBL::Hive::RunnableDB::SystemCmd \
    -input_id "{ 'cmd' => 'ls -l' }"
```

Parameter retrieval/storage API

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

- Jobs do not know where the parameters they are working with come from.
  All they need to know is:

  - How to get a value of a parameter:
    ```
    my $alpha = $self->param('alpha');
    ```
  - How to set it to make available to other parts of Job's lifecycle:
    ```
    $self->param('beta', $beta);
    ```
  - How to require that the given parameter has been passed:
    ```
    my $alpha = $self->param_required('alpha');
    ```
  - How to check whether it is defined:
    ```
    if ($self->param_is_defined( 'gamma' ) ) { … }
    ```

- Parameters that you have stored in $self->param() are not automatically dataflown anywhere, it is your responsibility to trigger Dataflow Events:

Dataflow API

- Dataflow event has two parameters: a hash and branch number:

```
$self->dataflow_output_id( { 'alpha' => 1.5, 'gamma' => 5 }, 3 );
```

- The first parameter can also be an arrayref (of hashes):

```
$self->dataflow_output_id( [{'name' => 'Alice'}, {'name' => 'Bob'}], 2);
```

- Feel free to use any number of distinct dataflow branches to create events, they do not have to be all wired. You can create different modes of operation by wiring different branches. A separate branch_number should be allocated for each distinct kind of data.

- Be careful when explicitly dataflowing into branch #1, as this will override the autoflow. You shoud know what you are doing (multiple events in branch #1 is a bad idea).

- If you do explicitly dataflow into branch #1, make sure this Dataflow Event happens after all Dataflow Events you envisage may constitute a semaphore group with funnel in branch #1.

Error reporting API

- You may leave a non-fatal human-readable message in log_message table:

  ```
  $self->warning( 'I got a strange feeling I am in an infinite loop…' );
  ```

  Do not mix it with “warn” whose output will go to wherever STDERR of the Job is

- Any fatal message will also be recorded in log_message:

  ```
  die 'all gone wrong';   # just the message
  $self->throw();      # with call stack trace (including Hive internal calls)
  ```

- However the same die/throw/croak/... calls can be used to mark the successful completion of a Job. In this case you have to first unset the incomplete flag.

  ```
  $self->input_job->incomplete( 0 );
  die 'all gone right'; # this message is still recorded
  ```

- Setting transient_error to 0 and then dying will prevent further attempts to retry the Job:

  ```
  $self->input_job->transient_error( 0 );
  if( $alpha < 0) { die “alpha parameter cannot be negative”; }
  $self->input_job->transient_error( 1 );
  ```

- You can also instruct the Worker to exit if you believe it has been contaminated:

  ```
   $self->input_job->lethal_for_worker( 1 );
  die “There is no point to carry on with this Worker: /tmp is full”;
  ```

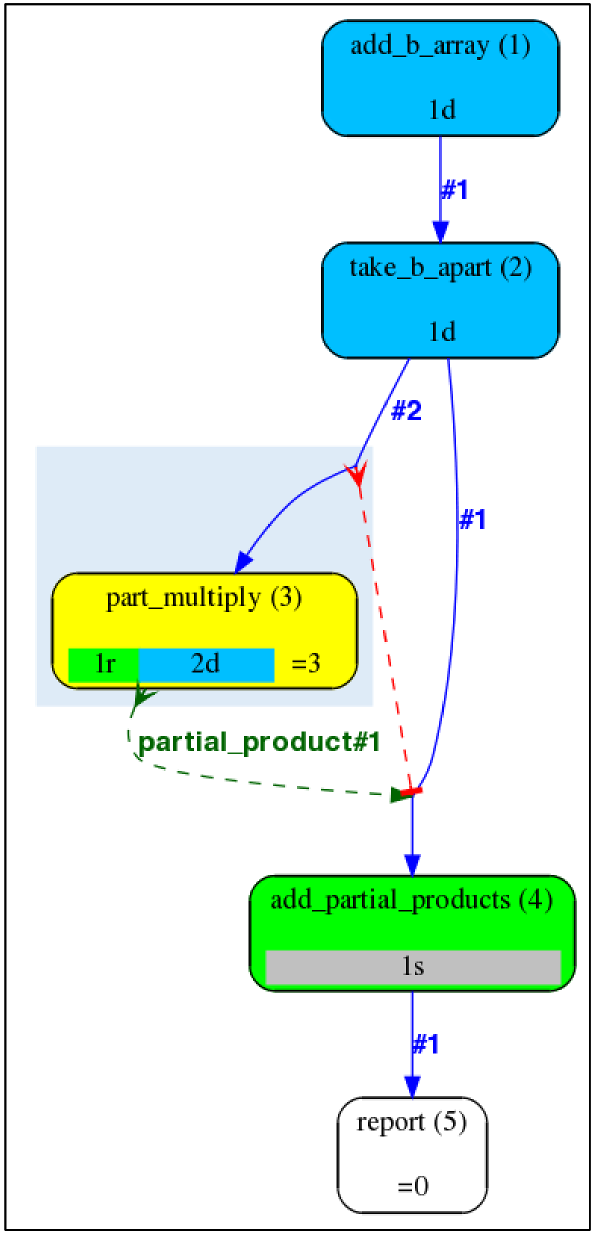
Exercise of the day: checking the distributive property

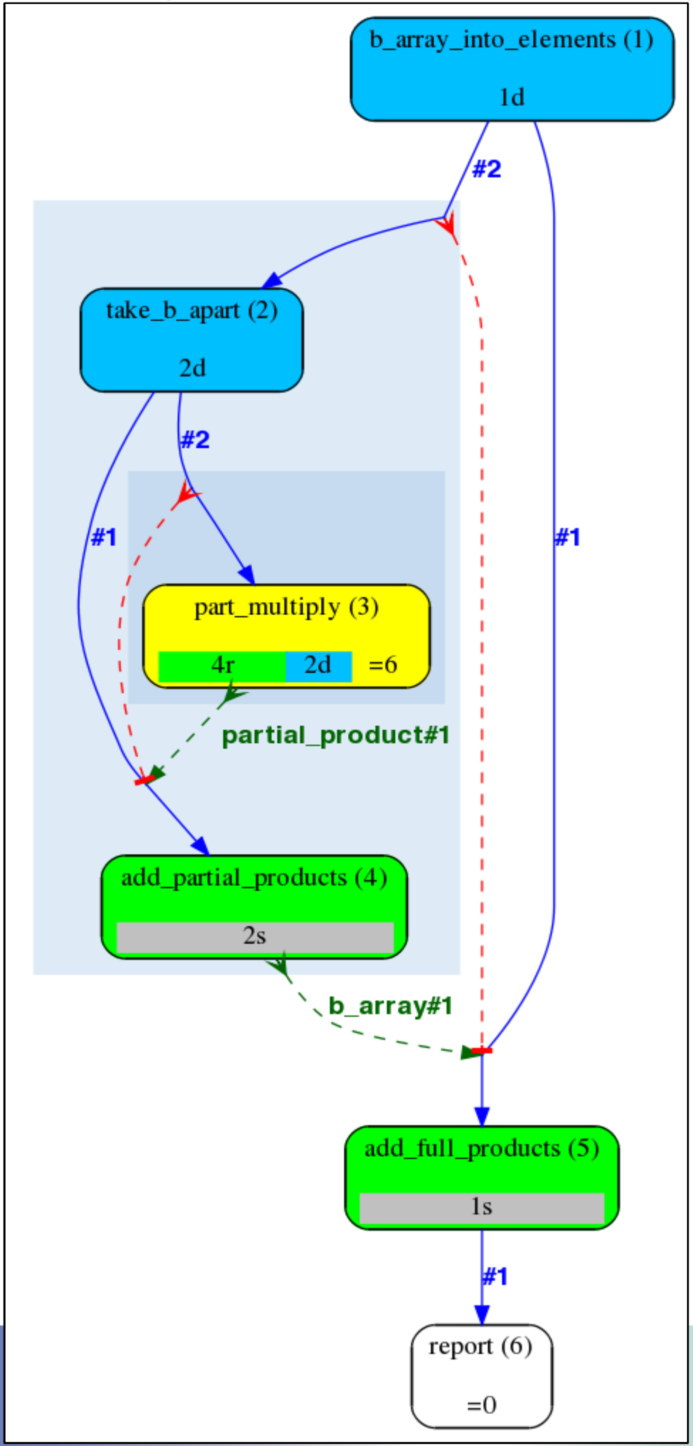
- Our main exercise will be to construct a pipeline that will check experimentally that A*(B1+B2+B3+...+Bn) == A*B1+A*B2+A*B3+...+A*Bn.

- Long Multiplication pipeline components (3 runnables and the configuration file) can be reused, but you will need to write our own AddArray.pm runnable for adding together array members.

- We will need to create two configuration files:
  - AddThenMultiply_conf.pm to define the flow of the left part
  - MultiplyThenAdd_conf.pm to define the flow of the right part

AddThenMultiply_conf vs MultiplyThenAdd_conf





- Interface to the Runnable: AddArray.pm should
  - take in 'b_array' parameter
  - dataflow the 'sum' parameter into branch #1
- Interface to the PipeConfigs: we want to seed both pipelines with the same input_id:

```
{
    'a_multiplier' => 123,
    'b_array' => [456,789]
}
```

- How can we keep the same interface in these two contexts?

Templates: the other kind of glue

- Runnables have *fixed parameter names* for input and output -
  in comparison with Perl subroutine calls that have a *fixed order of parameters*:
  - + 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 may 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#',          # explicit propagation
         },
       'another_analysis' => undef,   # no template - use as is
},
```

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

Now you should know everything you need...

- ... to finish the exercise.

- Solutions:
  - `AddThenMultiply1_conf.pm` and `MultiplyThenAdd1_conf.pm`
    (to test the structure of pipelines without an extra Runnable)
  - `AddThenMultiply2_conf.pm` , `MultiplyThenAdd2_conf.pm` and `AddArray.pm`
    `( with a dedicated Runnable )`

- Feel free to merge two parts into one – with an automatic comparator in the end.

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

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'    => '*',
        };
}
```

always inherit from the parent

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

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