THE GENOME INSTITUTE at Washington University

Ptero

Mark M. Burnett, Travis Abbott, Ian T. Ferguson,
Michael Kiwala, David Morton, Richard K. Wilson

genome.github.io/ptero

Purpose

To support large scale human genetics studies, it is necessary to execute workflows with tens of thousands of parallel operations. When it became clear that our existing workflow management system could not scale to this level, we investigated other existing systems. We concluded that that the combination of performance and a clear integration path with our deployed software did not exist, so we began development of a new workflow system.

Since the first version of PTero was deployed at TGI in the summer of 2013, it has orchestrated more than 540,000 workflows and over 8 million operations. While our typical workflows have only dozens of operations, the largest had 19,478 operations.

Architecture

PTero is composed of several services that provide REST APIs. Each service has a narrow scope of responsibility. The services are:

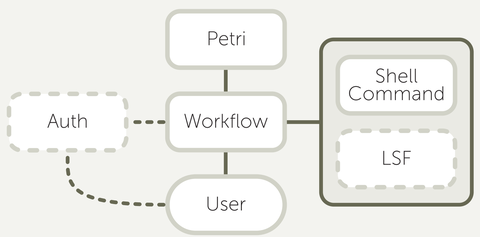
- **Workflow** — orchestrates inputs and outputs for tasks
- **Petri** — manages flow control
- **Auth** — manages user credentials
- **LSF** — submits jobs to an LSF cluster on behalf of the user
- **Shell Command** — executes local processes without resource management

The primary method for extending the system to is add new execution services that connect with back-ends such as SGE, EC2, or Nova.

To interact with the system, a user constructs a serialized representation of the workflow they wish to run. After authenticating with the Auth service, the representation is then submitted to the Workflow service, which validates the workflow and generates a Petri net corresponding to the flow control of the workflow. That Petri net is submitted to the Petri service, which executes it.

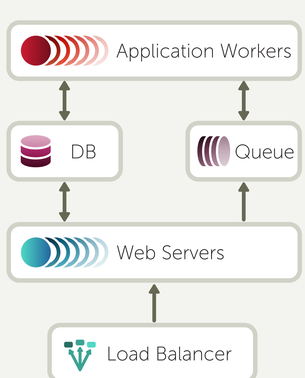
Petri notifies the Workflow service when jobs are ready to execute, and Workflow makes requests to appropriate execution services (e.g. Shell Command or LSF) to run the job with the correct inputs.

The execution services call back into Workflow with status updates and job outputs, which are stored by the Workflow service. When a job has finished, Workflow notifies the Petri service, so that it can manage further progress in the workflow's Petri net. Finally, Petri will notify the Workflow service when the net is complete.



Scalability

Scalability of each individual service allows the overall PTero system to scale. The services are implemented using web application architecture with load balancing and application workers listening on queues for work. Increasing application throughput is possible with this architecture by adding more web servers or application workers as needed.

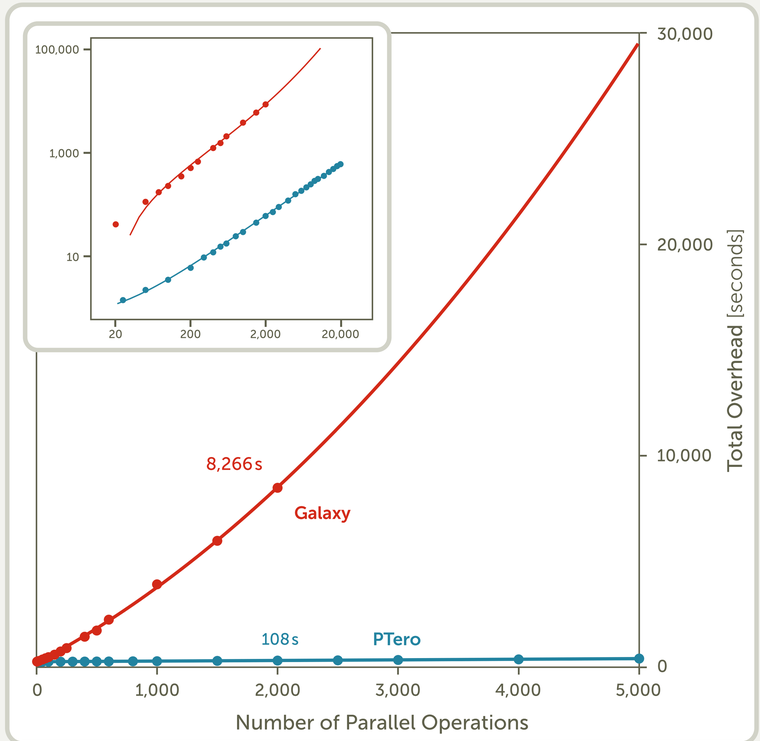


References

Amazon Web Services Reference Architecture.
http://media.amazonwebservices.com/architecturecenter/AWS_ac_ra_web_01.pdf

Murata, Tadao. Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE, Vol. 77, No. 4, April 1989.

Benchmarking



Measured overhead in seconds for parallel workflows as a function of the number of operations in each workflow. Points represent measurements, and lines represent polynomial fits to the data. (Inset) Log-log scale

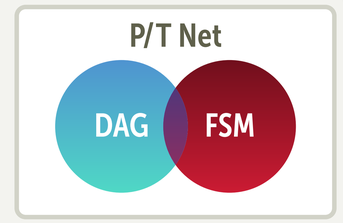
This figure shows the measurements of system overhead for workflows with a large number of parallel operations with individually predictable run times (each sleeping for 0.1 seconds). Overhead was calculated by measuring total run time for a workflow, then subtracting the theoretically fastest run time.

The main plot shows the total overheads directly, with a linear fit for the PTero data and a quadratic fit for the Galaxy data (both fits have $R^2 > 0.99$). The F-test comparing linear and quadratic fits to the Galaxy data suggests rejecting the linear model with confidence level of 99.9%. The quadratic model predicts a total overhead for a parallel workflow with 20,000 nodes at a little over 3 days.

Measurements for PTero and Galaxy were taken inside an Ubuntu 14.04 VM with 4 virtual CPUs and 4 GB of RAM, running in VirtualBox on 2013 MacBook Pro Retina 15" laptops. Each workflow system was running 4 workers performing the sleep operations.

The configuration details and code used to perform these benchmarks is available at **github.com/genome/ptero**.

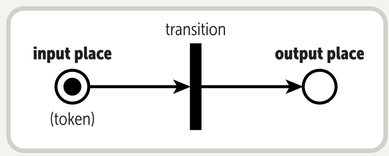
Petri Nets



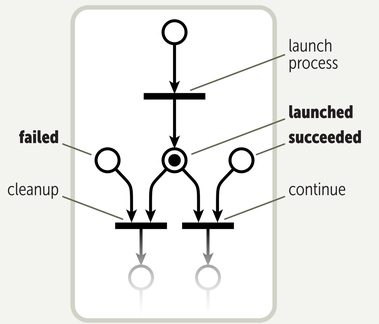
Workflow systems usually represent workflows as a set of tasks arranged in a directed acyclic graph (DAG). The tasks are often best modeled as being in one of many mutually exclusive states using a finite state machine (FSM). Petri nets, or P/T nets, are more generalized and can behave as DAGs or FSMs.

	Dependency	Concurrency	Choice
DAG	Yes	Yes	No
FSM	Yes	No	Yes
P/T	Yes	Yes	Yes

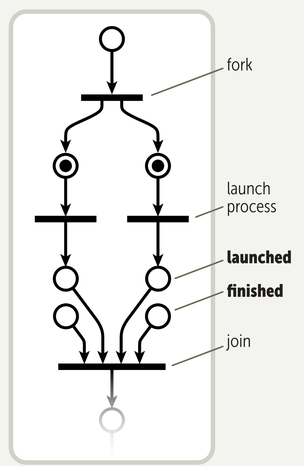
Both DAGs and FSMs are able to represent dependencies between tasks, or sequencing. But since FSMs are not able to produce concurrency and DAGs cannot model choice, neither is a sufficient representation for both tasks and workflows. Petri nets are capable of dependency, concurrency, and choice.



Petri nets are bipartite graphs with "transitions" representing events/processes and "places" representing states/conditions. A transition "fires" when all of its input-places (i.e. preconditions) contain a token. After firing, one token is placed into each output-place.



Choice



Concurrency

Workflow

The Workflow service allows users to specify a process using a Directed Acyclic Graph (DAG) based representation with additional features for organizing data flow, including:

- Efficient parallelization of nodes on the outputs of other nodes.
- Individual nodes may be lists of methods executed in order until one succeeds.
- Nodes may be DAGs themselves, allowing construction of large workflows by composing smaller ones.
- The service does not require registration of tools, so tools may be added or removed without service disruption.

Figure A shows the serialized representation of a workflow along with its DAG visualization. Because the input of node C is specified as **`parallelBy`**, node B will be parallelized across the output of node A and the outputs of the individual executions of node C will be collected into an array when passed into the input of node D.

Figure B shows the Petri net representation for node B of the workflow in Figure A, which first attempts one method of execution, then falls back on a second if the first method fails.

```
{
  "nodes": {
    "A": { ... },
    "B": {
      "methods": [{
          "service": "ShellCommand",
          "parameters": { "commandLine": ["/usr/bin/b", "checkpoint"] }
        },
        {
          "service": "LSF",
          "parameters": {
              "commandLine": ["/usr/bin/b", "execute"],
              "queue": "long",
              "resReq": "select[mem>8000] rusage[mem=8000] span[hosts=1]" }
        }]
    },
    "C": {
      "methods": [{
          "service": "ShellCommand",
          "parameters": { "commandLine": ["/usr/bin/c"] }
      }],
      "parallelBy": "result_of_a"
    },
    "D": { ... }
  },
  "edges": [
    {
      "source": "B",
      "destination": "D",
      "sourceProperty": "result_of_b",
      "destinationProperty": "input_1_of_d"
    },
    ...
  ],
  "inputs": { ... }
}
```

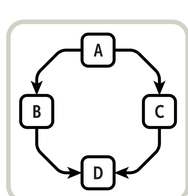


Figure A

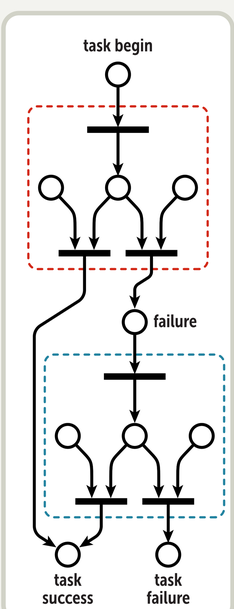


Figure B