Python para HPC

Professores: Guilherme Gall, Luiz Gadelha

Escola do Supercomputador Santos Dumont
Programa de Verão de 2019
Laboratório Nacional de Computação Científica

22 de fevereiro de 2019

Modelos de Programação Paralela

- Modelos puros de programação paralela:
  - Memória compartilhada:
    - Características: para sistemas de memória compartilhada, alternância entre seções de código sequenciais e paralelas (fork/join).
    - Exemplo: OpenMP.
  - Passagem de mensagens:
    - Características: para sistemas de memória compartilhada ou distribuída, diversos processos trocam mensagens.
    - Exemplo: MPI.

Diaz, J. et al. (2012). A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. IEEE Transactions on Parallel and Distributed Systems, 23(8), 1369–1386. https://doi.org/10.1109/TPDS.2011.308

Modelos de Programação Paralela

- Modelo de programação paralela heterogênea:
  - Características: para sistemas de que possuem aceleradores (p.ex. GPUs), além das CPUs.
  - Exemplos: CUDA, OpenCL.
- **Workflows científicos**:
  - Características: para sistemas de memória compartilhada ou distribuída, tarefas independentes processam conjuntos de dados de forma encadeada (*dataflow*).
  - Exemplos: Pegasus, Swift, **Parsl**.

Programming in the small × *programming in the large*

- *Programming in the small*:
  - Desenvolvimento de módulos relativamente "pequenos", com escopo bem-definido, como aplicações científicas e conversores de formato de dados.
  - Geralmente utiliza linguagens como C/C++, Fortran e R.
- *Programming in the large*:
  - Utilização de linguagens para interconexão dos módulos, que especificam a interação entre os mesmos para a construção de sistemas mais complexos.
  - Caso típico de utilização de sistemas de gerenciamento de workflows científicos, como o Swift.

De Remer, F., Kron, H. (1975). Programming-in-the large versus programming-in-the-small. ACM SIGPLAN Notices, 10(6), 114–121.

Workflows Científicos

- Um **workflow científico** consiste da especificação de um encadeamento de aplicações científicas a serem executadas e de dependências mútuas.
- Segue um ciclo de vida análogo ao dos experimentos científicos computacionais:
  - Composição, representação e modelagem de dados.
  - Mapeamento e execução.
  - Coleta de metadados e proveniência.
- Um **sistema de gerência de workflows científicos** (SGWC) permite gerenciar o ciclo de vida de workflows científicos.

Liu, J., Pacitti, E., Valduriez, P., Mattoso, M. (2015). A Survey of Data-Intensive Scientific Workflow Management. Journal of Grid Computing, 13(4), 457–493.

Workflows Científicos

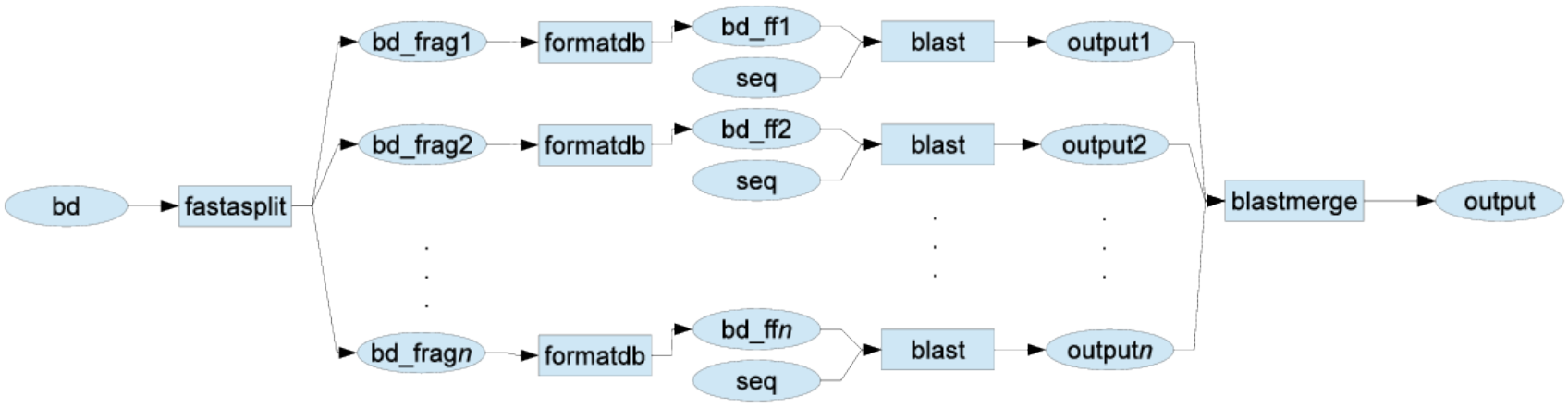
- **Sistemas de gerência de workflows científicos** (SGWC) visam a automação de experimentos científicos computacionais:
  - escalonamento de tarefas baseado em dependências de dados;
  - fluxo de dados entre tarefas;
  - execução paralela de tarefas independentes;
  - escalonamento de tarefas em ambientes de computação de alto desempenho;
  - gerência e consulta de dados de proveniência.

Exemplo: Laboratório de Bioinformática, LNCC

Sequenciador Roche 454 FLX gera 12GB a 15GB por rodada de sequenciamento.
Dados processados por várias aplicações:

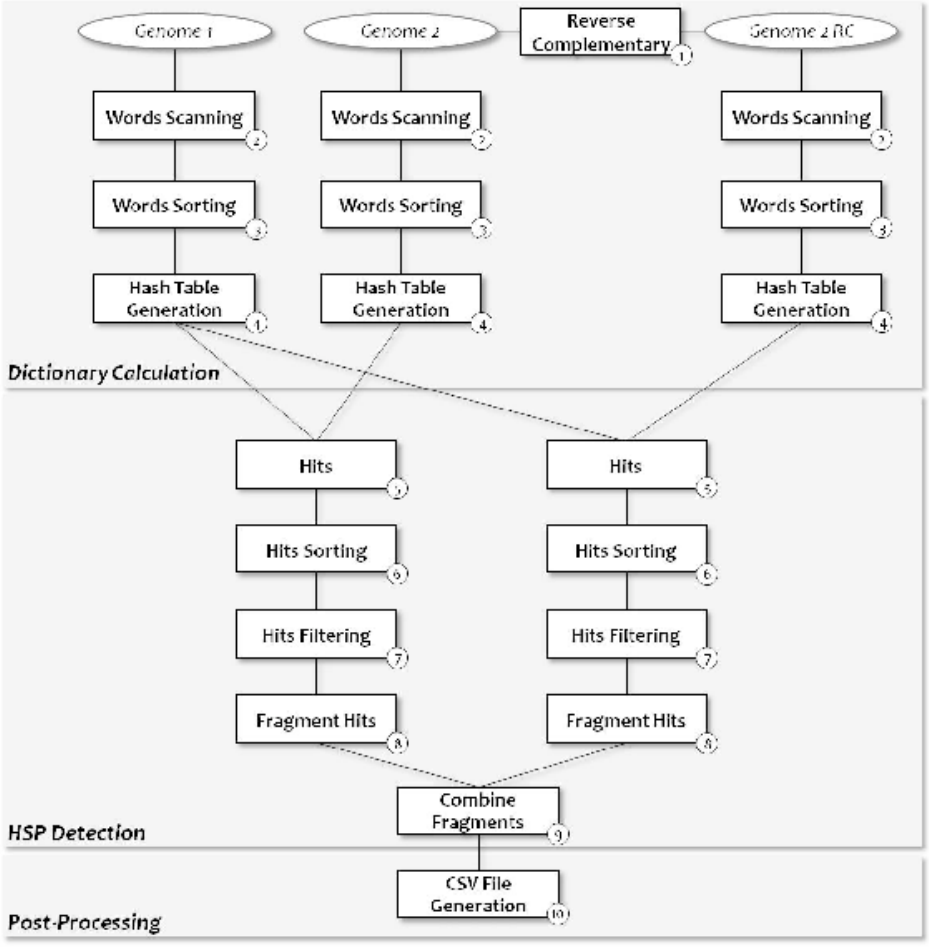
- filtragem por qualidade de dados gerados pelo sequenciador.
- formatação de dados.
- comparação das sequências com bases de dados conhecidas.

Exemplo: BLAST paralelo



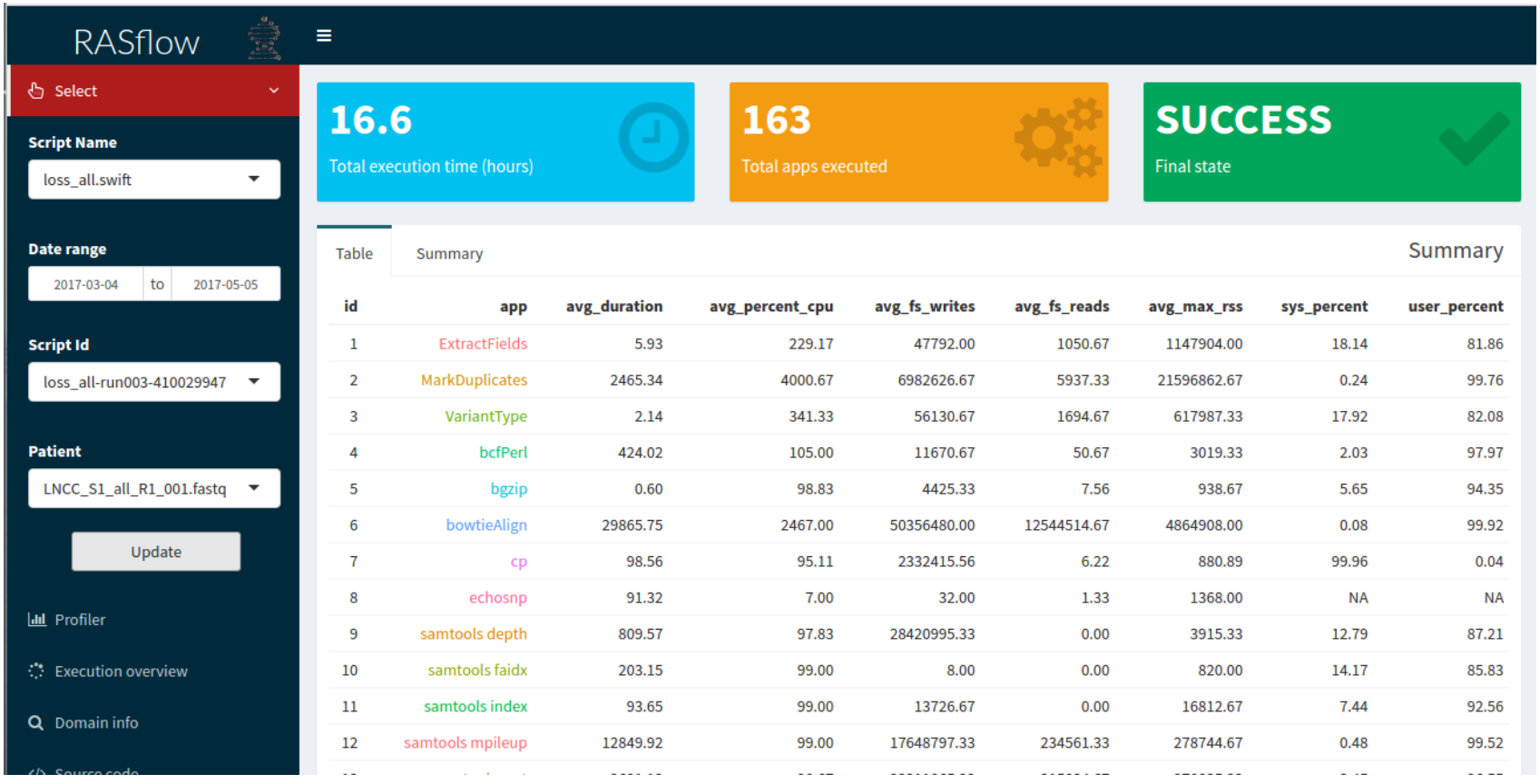
https://github.com/lgadelha/swift-blast

BioWorkbench



Mondelli, M. L. et al. (2018). BioWorkbench: a high-performance framework for managing and analyzing bioinformatics experiments. PeerJ, 6:e5551. https://doi.org/10.7717/peerj.5551

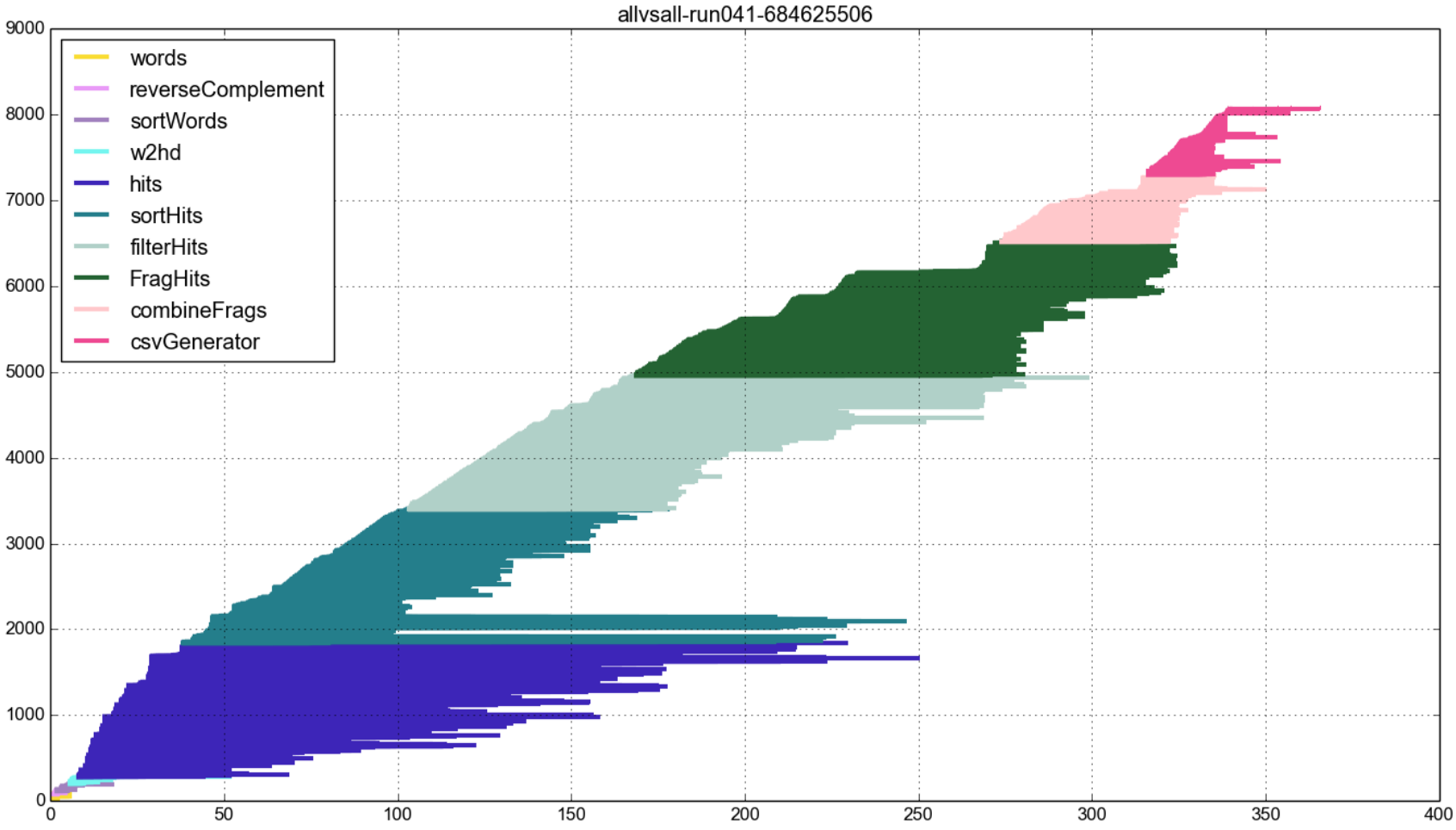
BioWorkbench



id	app	avg_duration	avg_percent_cpu	avg_fs_writes	avg_fs_reads	avg_max_rss	sys_percent	user_percent
1	ExtractFields	5.93	229.17	47792.00	1050.67	1147904.00	18.14	81.86
2	MarkDuplicates	2465.34	4000.67	6982626.67	5937.33	21596862.67	0.24	99.76
3	VariantType	2.14	341.33	56130.67	1694.67	617987.33	17.92	82.08
4	bcfPerl	424.02	105.00	11670.67	50.67	3019.33	2.03	97.97
5	bgzip	0.60	98.83	4425.33	7.56	938.67	5.65	94.35
6	bowtieAlign	29865.75	2467.00	50356480.00	12544514.67	4864908.00	0.08	99.92
7	cp	98.56	95.11	2332415.56	6.22	880.89	99.96	0.04
8	echosnp	91.32	7.00	32.00	1.33	1368.00	NA	NA
9	samtools depth	809.57	97.83	28420995.33	0.00	3915.33	12.79	87.21
10	samtools faidx	203.15	99.00	8.00	0.00	820.00	14.17	85.83
11	samtools index	93.65	99.00	13726.67	0.00	16812.67	7.44	92.56
12	samtools mpileup	12849.92	99.00	17648797.33	234561.33	278744.67	0.48	99.52

Mondelli, M. L. et al. (2018). BioWorkbench: a high-performance framework for managing and analyzing bioinformatics experiments. PeerJ, 6:e5551. https://doi.org/10.7717/peerj.5551

BioWorkbench



Mondelli, M. L. et al. (2018). BioWorkbench: a high-performance framework for managing and analyzing bioinformatics experiments. PeerJ, 6:e5551. https://doi.org/10.7717/peerj.5551

Workflows Científicos

- Workflows científicos típicos:
  - *Bag of tasks* (MG-RAST, DOCK),
  - Múltiplos estágios que usam arquivos como dados intermediários (Montage, BLAST),
  - Aplicações distribuídas com pares chave-valor intermediários (histogramas de dados para física de altas energias),
  - Cadeias de tarefas do tipo MapReduce,
  - Aplicações iterativas com variação no número de tarefas (otimização),

Deelman, E. et al. (2018). The future of scientific workflows. The International Journal of High Performance Computing Applications, 32(1), 159–175. https://doi.org/10.1177/1094342017704893

Exemplo de paralelização do BLAST com Swift

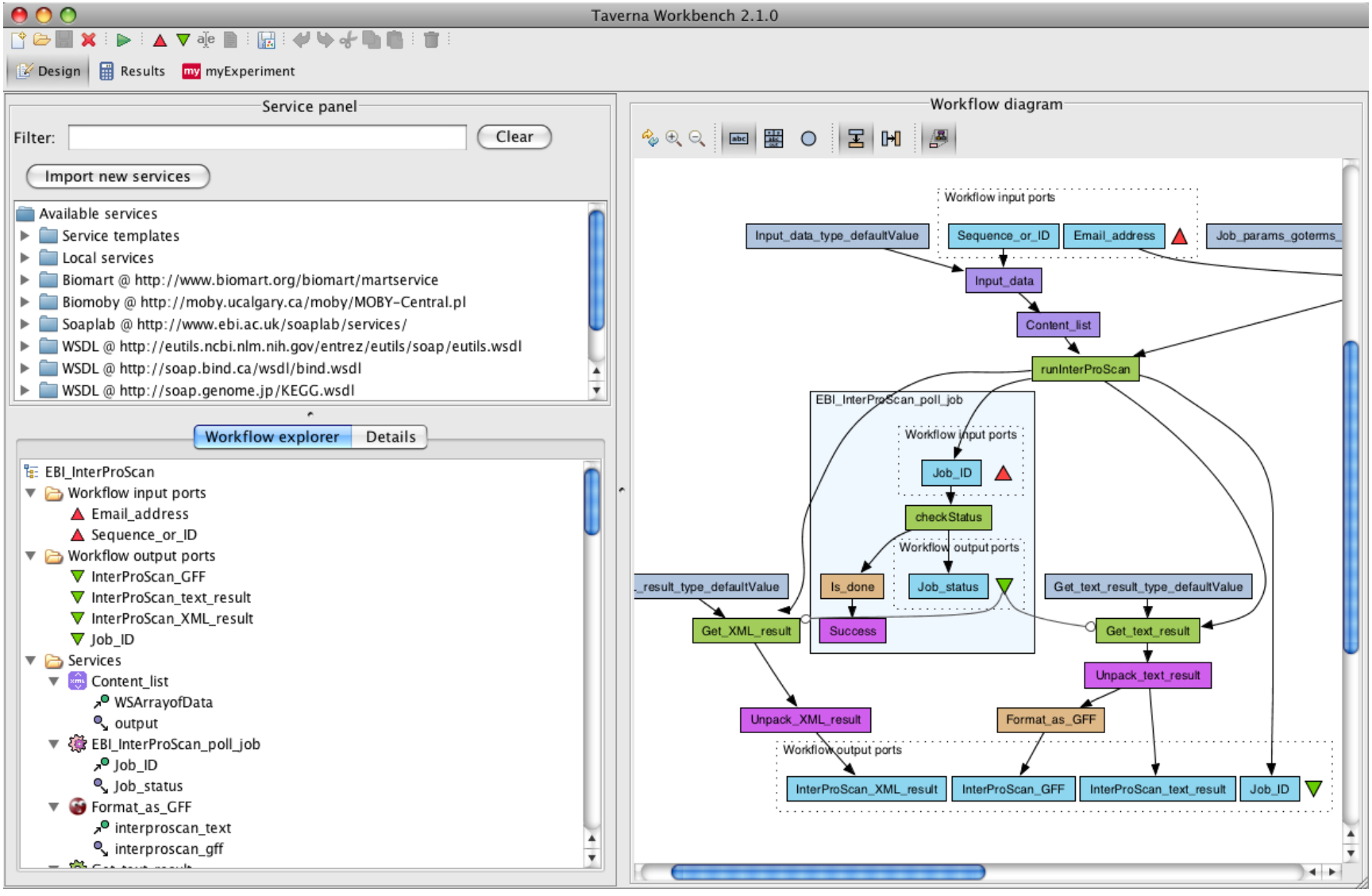
```
partition=split_database(dbin, num_partitions);

database formatdbout[] <ext; exec="formatmapper.sh",n=num_partitions>;
output out[] <ext; exec="outmapper.sh",n=num_partitions>;

foreach part,i in partition {
  formatdbout[i] = formatdb(part);
  out[i]=blastapp(query_file, part, program_name, expectation_value,
                  filter_query_sequence, formatdbout[i]);
}

blast_output_file=blastmerge(out);
```

Exemplo: Composição Gráfica

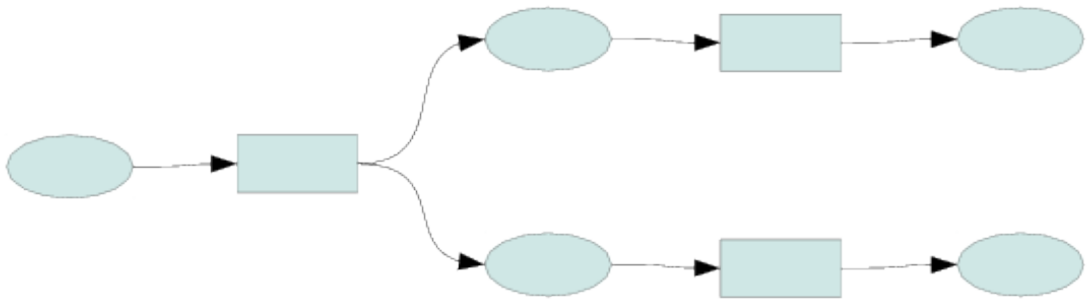


Fonte: Taverna (`http://www.taverna.org.uk`)

Composição de Workflows Científicos: Padrões

- Foram identificados 43 padrões de composição de workflows.
- Exemplos:
  - Seqüência:

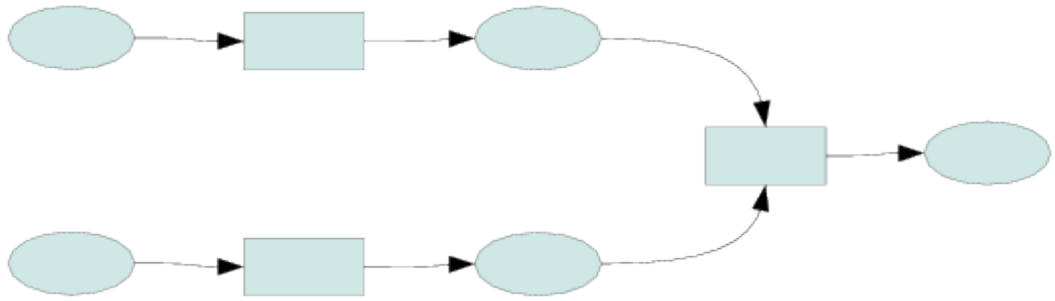
  - Bifurcação paralela:



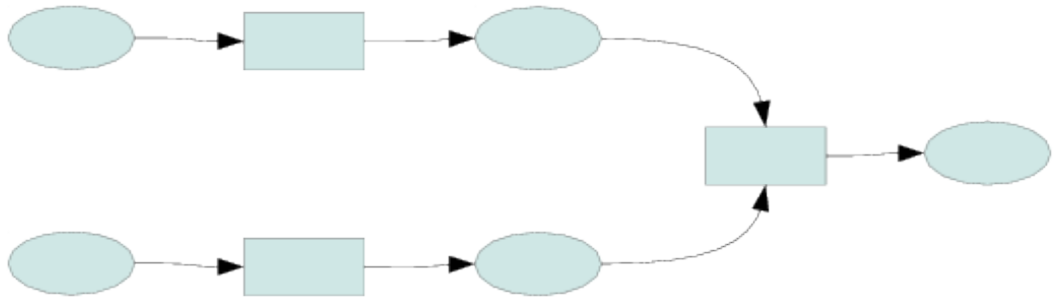
- W. van der Aalst et al. Workflow Patterns. *Distributed and Parallel Databases* 14(1):5-51, 2003.
- N. Russell et al. Workflow Control-Flow Patterns: A Revised View. BPM Center Report BPM-06-22, 2006.
- Workflow Patterns (http://www.workflowpatterns.com).

Composição de Workflows Científicos: Padrões

- Exemplos:
  - Sincronização:

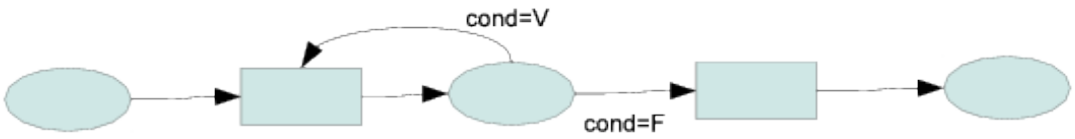


  - Fusão:

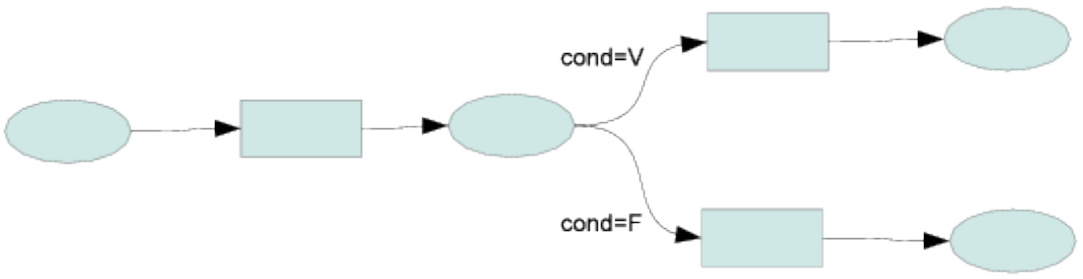


Composição de Workflows Científicos: Padrões

- Exemplos:
  - Laço:



  - Escolha exclusiva:



Histórico do Parsl

- Versões do Swift:
  - Swift/K (2007):
    - workflows paralelos e distribuídos.
    - `http://swift-lang.org`.
  - Swift/T (2013):
    - workflows para HPC (p.ex. máquinas Top 500).
    - `http://swift-lang.org/Swift-T`.
  - Parsl (2015):
    - biblioteca do Python que incorpora ambiente de execução do Swift.
    - workflow passa a ser especificado em Python.
    - `http://parsl-project.org/`.

Armstrong, T. et al. (2015). Swift: Extreme-scale, Implicitly Parallel Scripting. In P. Balaji (Ed.), Programming Models for Parallel Computing (pp. 219–245). MIT Press.
`https://www.researchgate.net/publication/284444451_Swift_Extreme-scale_Implicitly_Parallel_Scripting`.

Swift

- Desenvolvido no *Argonne National Laboratory* e *University of Chicago*
- Há uma colaboração com o LNCC no desenvolvimento do componente de gerência de proveniência.
- Exemplos de workflows científicos implementados no LNCC:
  - swift-blast (alinhamento paralelo de sequências).

    `https://github.com/lgadelha/swift-blast`
  - SwiftGECKO (genômica comparativa).

    `https://github.com/mmondelli/swift-gecko`
  - sdm-workflows (modelagem de distribuição de espécies).

    `https://github.com/sibbr/sdm-workflows`

Parsl

- Parsl é uma biblioteca para o Python que permite gerenciar workflows científicos paralelos.
- O paralelismo é implícito e baseado nas dependências das aplicações componentes do workflow.
- O motor de execução inclui diversas opções de execução, desde multithreading local a execução em supercomputadores com Swift/T.

`http://parsl-project.org/`.

Babuji, Y. et al. (2018). Parsl : Scalable Parallel Scripting in Python. In 10th International Workshop on Science Gateways (IWSG 2018).

Instalação do Parsl

- Tutorial baseado em `https://parsl.readthedocs.io`
- Parsl requer Python $\geq$ 3.5.

```
$ pip3 install parsl
$ pip3 install jupyter
```

Verificando a versão do Parsl

```
$ python3
>>> import parsl
>>> from parsl.app.app import python_app, bash_app
>>> from parsl.configs.local_threads import config
>>> print(parsl.__version__)

0.7.1
```

Carregando a configuração

- O Parsl permite expressar como o workflow é composto.
- Os detalhes de tempo de execução devem ser carregados de um arquivo de configuração.

```
>>> parsl.load(config)

<parsl.dataflow.dflow.DataFlowKernel object at 0x1029657b8>
```

Apps

- Uma App é um trecho de código que pode ser executado de forma assíncrona em algum recurso computacional (p.ex. nuvem, cluster ou PC).
- Parsl provê suporte aos seguintes tipos de Apps:
  - código nativo em Python (`python_app`),
  - programas executados via de linha de comando (`bash_app`).

Apps Python

- Uma App do tipo Python é declarado usando o decorador `@python_app`.

```
@python_app
def hello ():
    return 'Hello World!'

print(hello().result())
```

```
@python_app
def multiply (a, b):
    return a * b

print(multiply(5,9).result())
```

Apps Python

- Apps Python podem ser executadas em recursos remotos, as dependências devem ser carregadas na definição da App.

```
@python_app
def slow_hello ():
    import time
    time.sleep(5)
    return 'Hello World!'

print(slow_hello().result())
```

Apps Bash

- Uma App do tipo Python é declarado usando o decorador `@bash_app`.

```
@bash_app
def echo_hello(stdout='echo-hello.stdout',
               stderr='echo-hello.stderr'):
    return 'echo "Hello World!"'

echo_hello().result()

with open('echo-hello.stdout', 'r') as f:
     print(f.read())
```

Passando dados para as Apps

- As palavras-chaves `inputs` e `outputs` podem ser usadas nas declarações das Apps para expressar as dependências de dados.
- Vamos supor que geramos três arquivos de texto:

```
$ echo "hello 1" > /tmp/hello1.txt
$ echo "hello 2" > /tmp/hello2.txt
$ echo "hello 3" > /tmp/hello3.txt
```

Passando dados para as Apps

- Eles podem ser processados pelo App cat da seguinte forma:

```
@bash_app
def cat(inputs=[], outputs=[]):
    return 'cat %s > %s' %(" ".join(inputs), outputs[0])

concat = cat(inputs=['/tmp/hello1.txt','/tmp/hello2.txt','/tmp/
    hello3.txt'],
             outputs=['all_hellos.txt'])

# Open the concatenated file
with open(concat.outputs[0].result(), 'r') as f:
     print(f.read())
```

AppFutures

- Cada chamada a uma App retorna um *futuro*, que representa a execução assíncrona da App:

```
@python_app
def hello ():
    import time
    time.sleep(20)
    return 'Hello World!'

app_future = hello()

# Check if the app_future is resolved
print ('Done: %s' % app_future.done())

# Print the result of the app_future.
print ('Result: %s' % app_future.result())
print ('Done: %s' % app_future.done())
```

DataFutures

- Cada chamada a uma App deve especificar `outputs`, que são DataFutures.
- DataFutures são monitorados para garantir que sejam criados e passados para Apps que dependem deles.

```
@bash_app
def slowecho(message, outputs=[]):
    return 'sleep 5; echo %s &> {outputs[0]}' % (message)

hello = slowecho('Hello World!',
        outputs=[os.path.join(os.getcwd(), 'hello-world.txt')])

print(hello.outputs)
[<DataFuture at 0x10e0b29b0 state=running>]

print(hello.outputs)
[<DataFuture at ... state=finished returned hello-world.txt>]
```

DataFutures

```
hello2 = slowecho('Hello World!', outputs=[os.path.join(os.
    getcwd(), 'hello-world2.txt')])

print ('Done: %s' % hello.done())
Done: False

with open(hello2.outputs[0].result(), 'r') as f:
     print(f.read())
Hello World!

print(hello2.outputs)
[<DataFuture at ... state=finished returned hello-world2.txt>]

print ('Done: %s' % hello2.done())
Done: True
```

Cálculo de π com Monte Carlo

- Estimar π jogando dardos em um quadrado unitário.
- Calcular o percentual que acerta um círculo unitário:
  - Área do quadrado: $r^2 = 1$.
  - Área do círculo no quadrante: $\frac{\pi r^2}{4} = \frac{\pi}{4}$.
- Jogando dardos randomicamente em posições x, y.
- Se $x^2 + y^2 < 1$, o ponto estará dentro do círculo.
- A proporção que acertará dentro ($\frac{\text{acertos}}{\text{tentativas}}$) esperada é $\frac{\text{acertos}}{\text{tentativas}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$.
- Logo $\pi = 4\frac{\text{acertos}}{\text{tentativas}}$

Exemplo de workflow Parsl

```
@python_app
def pi(total):
    import random
    width = 10000
    center = width/2
    c2  = center**2
    count = 0
    for i in range(total):
        # Drop a random point in the box.
        x,y = random.randint(1, width),
              random.randint(1, width)
        # Count points within the circle
        if (x-center)**2 + (y-center)**2 < c2:
            count += 1
    return (count*4/total)
```

Exemplo de workflow Parsl

```
@python_app
def mysum(a,b,c):
    return (a+b+c)/3

a, b, c = pi(10**6), pi(10**6), pi(10**6)
avg_pi  = mysum(a, b, c)
```

Configuração

- Configuração com múltiplas *threads* locais:

```
from parsl.config import Config
from parsl.executors.threads import ThreadPoolExecutor

local_threads = Config(
    executors=[
        ThreadPoolExecutor(
            max_threads=8,
            label='local_threads'
        )
    ]
)
```

Configuração

- Configuração com *job* piloto:

```
from parsl.providers import LocalProvider
from parsl.channels import LocalChannel
from parsl.config import Config
from parsl.executors import HighThroughputExecutor
local_htex = Config(
    executors=[
        HighThroughputExecutor(
            label="htex_Local",
            worker_debug=True,
            cores_per_worker=1,
            provider=LocalProvider(
                channel=LocalChannel(),
                init_blocks=1,
                max_blocks=1, ), ) ],
    strategy=None, )
```

Configuração

- Configuração para acesso remoto ao SLURM:

```
from parsl.providers import SlurmProvider
from parsl.channels import SSHChannel
from parsl.launchers import SrunLauncher
...
config = Config(
    executors=[
        IPyParallelExecutor(
            ...
            provider=SlurmProvider(
                'debug',
                channel=SSHChannel(
                    hostname='login.sdumont.lncc.br',
                    username='lgadelha',
                    ...
```

Parsl no Santos Dumont

```
$ module load python/3.5.2
$ python3
>>> import parsl
>>> from parsl.app.app import python_app
>>> from parsl.configs.local_threads import config
>>> parsl.load(config)
<parsl.dataflow.dflow.DataFlowKernel object at 0x7f69b47a9358>
>>> @python_app
>>>  def hello():
>>>      return "Hello world!"
>>>
>>> print(hello().result())
Hello world!
>>> print(parsl.__version__)
0.6.1
```

Obrigado!

E-mail: lgadelha@lncc.br
Página web: http://www.lncc.br/~lgadelha