ParallelQueue

Release 1.0.0

Aaron Janeiro Stone

Jul 01, 2021

CONTENTS:

A SimPy Extension for parallel queueing systems and routing.

CHAPTER ONE

SIMPY INTRODUCTION

SimPy is a Discrete Event Simulation (DES) package for Python. DES refers to a simulation system which will periodically introduce a specified event as it runs, discrete insofar as each event can be thought of as “happened” or “not happened (yet, or ever)”.

In SimPy, the DES system is contained within its Environment class. The user specifies a system which will run for the Environment upon executing `Environment.run()`.

Within the Environment , the user can define Process and Resource objects which will dynamically interact as the DES is run. In particular, Process objects are the generators of the discrete events we will be working with while a Resource defines an object to be interacted with in this DES “universe”.

In Python, generator objects can be thought of as distinct from functions. Whereas functions have the form of (for some manipulation, `foo(a)`, like `a+2`):

```
def function(a):
    return foo(a)
```

generators have the form of:

```
def generator(a):
    yield foo(a)
```

where `yield` instead denotes a single output which will not cause the generator itself to stop being considered. To make this clear, it is possible to have:

```
def generator(a):
    yield foo(a)
    print('Wait, I forgot this!')
    yield bar(a)
```

which is useful for an object which will be existing in our universe for a period of time. Return would instead immediately end the function and output `foo(a)`. For a more concrete example, consider:

```
def generator(a):
     yield print(a)
     print('Wait, I forgot this!')
     yield print(a + 1)
 gen = generator(1)
```

which, when evaluating with the `next` command, gives us:

```
>>> next(gen)
1
```

(continues on next page)

(continued from previous page)

```
>>> next(gen)
Wait, I forgot this!
2
```

In the end, we now have a way to progress an object in some form of time by taking “steps”.

CHAPTER TWO

MODULE DOCUMENTATION

2.1 Model Components

Basic building components (generators/processes) for parallel models. In general, the framework allows one to build a model by defining an arrival, routing, and job/servicing process such that work is introduced in the order of Arrivals->Router->Job/Servicing.

`class parallelqueue.network.Network(**kwargs)`

The Network constructor allows a user to create a queueing network by overriding each member. Upon generation, jobs flow through a network in the order of: Arrivals → Router → Job. By default, the Network class can be used to handle JSQd, Redundancy-d, and Threshold-(d,r) models with general arrival and service distributions.

`Arrivals(system, env, number, queues, **kwargs)`

This generator/process defines how jobs enter the network

`static Job(system, env, name, arrive, queues, choice, **kwargs)`

This generator/process defines the behaviour of a job (replica or original) after routing.

`Router(system, env, name, queues, **kwargs)`

This generator/process specifies the scheduling system used.

`parallelqueue.arrivals.DefaultArrivals(router, system, env, number, queues, **kwargs)`

General arrival process; interarrival times are defined by the given distribution

Parameters

- **router** – Router process.
- **system** (`base_models.ParallelQueueSystem`) – System providing environment.
- **env** (*simpy.Environment*) – Environment for the simulation
- **number** (*int*) – Max numberJobs of jobs if infiniteJobs is false.
- **queues** (*List[simpy.Resource]*) – A list of all queues making up the parallel system.

`parallelqueue.routers.DefaultRouter(job, system, env, name, queues, **kwargs)`

Specifies the scheduling system used. If replication is enabled, this router tracks each set of replicas using a `base_models.ParallelQueueSystem.ReplicaDict` which can be accessed by `network.Network.Job` processes.

Parameters

- **job** – Job process.
- **system** (`base_models.ParallelQueueSystem`) – System providing environment.
- **env** (*simpy.Environment*) – Environment for the simulation.
- **name** (*str*) – Identifier for the job.

- **queues** (*List[simpy.Resource]*) – A list of queues to consider.

parallelqueue.routers.**NoInSystem**(*R*)

Total number of Jobs in the resource R.

parallelqueue.routers.**QueueSelector**(*d*, *parallelism*, *counters*)

The actual queue selection logic.

parallelqueue.jobs.**DefaultJob**(*system*, *env*, *name*, *arrive*, *queues*, *choice*, ***kwargs*)

For a redundancy model, this generator/process defines the behaviour of a job (replica or original) after routing.

Parameters

- **system** (base_models.ParallelQueueSystem) – System providing environment.
- **env** (*simpy.Environment*) – Environment for the simulation
- **name** (*str*) – Identifier for the job.
- **queues** (*List[simpy.Resource]*) – A list of queuesOverTime.
- **arrive** (*float*) – Time of job arrival (before replication).
- **choice** (*int*) – The queue which this replica is currently in

2.2 Standard Parallelization Models

Generators and simulation environments included under *base_models* focus on the modelling of JSQ(d), Redundancy-d and Threshold-d routing schemes based on queue size.

parallelqueue.base_models.**JSQd**(*parallelism, seed, d, Arrival, AArgs, Service, SArgs, Monitors=[<class 'parallelqueue.monitors.TimeQueueSize'>], r=None, maxTime=None, doPrint=False, infiniteJobs=True, numberJobs=0*)

A queueing system wherein a Router chooses the smallest queue of d sampled (identical) queues to join for each arriving job.

Parameters

- **maxTime** – If set, becomes the maximum allotted time for this simulation.
- **numberJobs** – Max number of jobs if infiniteJobs is False. Will be ignored if infiniteJobs is True.
- **parallelism** – Number of queues in parallel.
- **seed** – Random number generation seed.
- **r** – Threshold. Should be set to an integer, defaulting to None otherwise.
- **infiniteJobs** – If true, there will be no upper limit for the number of jobs generated.
- **d** – Number of queues to parse.
- **doPrint** – If true, each event will trigger a statement to be printed.
- **Arrival** – A kwarg specifying the arrival distribution to use (a function).
- **AArgs** – parameters needed by the function.
- **Service** – A kwarg specifying the service distribution to use (a function).
- **SArgs** – parameters needed by the function.
- **Monitors** – List of monitors which overrides the methods of monitors.Monitor

class parallelqueue.base_models.**ParallelQueueSystem**(*parallelism*, *seed*, *d*, *r=None*, *maxTime=None*, *doPrint=False*, *infiniteJobs=True*, *Replicas=True*, *numberJobs=0*, *network=<class 'parallelqueue.network.Network'>*, ***kwargs*)

A queueing system wherein a Router chooses the smallest queue of d sampled (identical) queues to join, potentially replicating itself before enqueueing. For the sampled queues with sizes less than r, the job and/or its clones will join while awaiting service. After completing service, each job and its replicas are disposed of.

Parameters

- **maxTime** – If set, becomes the maximum allotted time for this simulation.
- **numberJobs** – Max number of jobs if infiniteJobs is False. Will override infiniteJobs if infiniteJobs is True.
- **parallelism** – Number of queues in parallel.
- **seed** – Random number generation seed.
- **r** – Threshold. Should be set to an integer, defaulting to `None` otherwise.
- **infiniteJobs** – If true, there will be no upper limit for the number of jobs generated.
- **df** – Whether or not a pandas.DataFrame of the queue sizes over time should be returned.
- **d** – Number of queues to parse.
- **doPrint** – If true, each event will trigger a statement to be printed.
- **Arrival** – A kwarg specifying the arrival distribution to use (a function).
- **AArgs** – parameters needed by the function.
- **Service** – A kwarg specifying the service distribution to use (a function).
- **SArgs** – parameters needed by the function.
- **Monitors** – Any monitor which overrides the methods of monitors.Monitor
- **Network** – Network class which defines the structure of the system.

Example

```
# Specifies a SimPy environment consisting
# of a Redundancy-2 queueing system and a Poisson arrival process.
sim = ParallelQueueSystem(maxTime=100.0,
    parallelism=100, seed=1234, d=2, Replicas=True,
    Arrival=random.expovariate, AArgs=0.5,
    Service=random.expovariate, SArgs=1)
sim.RunSim()
```

References

Heavy Traffic Analysis of the Mean Response Time for Load Balancing Policies in the Mean Field Regime
Tim Hellemans, Benny Van Houdt (2020) https://arxiv.org/abs/2004.00876

Redundancy-d:The Power of d Choices for Redundancy Kristen Gardner, Mor Harchol-Balter, Alan Scheller-Wolf, Mark Velednitsky, Samuel Zbarsky (2017) https://doi.org/10.1287/opre.2016.1582

`property DataFrame`

If `TimeQueueSize` was a monitor, returns a dataframe of queue sizes over time.

`property MonitorOutput`

The data acquired by the monitors as observed during the simulation.

`RunSim()`

Runs the simulation.

`parallelqueue.base_models.RedundancyQueueSystem`(*parallelism, seed, d, Arrival, AArgs, Service, SArgs, Monitors=[<class 'parallelqueue.monitors.TimeQueueSize'>], r=None, maxTime=None, doPrint=False, infiniteJobs=True, numberJobs=0*)

A queueing system wherein a Router chooses the smallest queue of d sampled (identical) queues to join, potentially replicating itself before enqueueing. For the sampled queues with sizes less than r, the job and/or its clones will join while awaiting service. After completing service, each job and its replicas are disposed of.

Parameters

- **`maxTime`** – If set, becomes the maximum allotted time for this simulation.
- **`numberJobs`** – Max number of jobs if infiniteJobs is False. Will be ignored if infiniteJobs is True.
- **`parallelism`** – Number of queues in parallel.
- **`seed`** – Random number generation seed.
- **`r`** – Threshold. Should be set to an integer, defaulting to `None` otherwise.
- **`infiniteJobs`** – If true, there will be no upper limit for the number of jobs generated.
- **`d`** – Number of queues to parse.
- **`doPrint`** – If true, each event will trigger a statement to be printed.
- **`Arrival`** – A kwarg specifying the arrival distribution to use (a function).
- **`AArgs`** – parameters needed by the function.
- **`Service`** – A kwarg specifying the service distribution to use (a function).
- **`SArgs`** – parameters needed by the function.
- **`Monitors`** – List of monitors which overrides the methods of monitors.Monitor

Example

```
# Specifies a SimPy environment consisting
# of a Redundancy-2 queueing system and a Poisson arrival process.
sim = RedundancyQueueSystem(maxTime=100.0,
    parallelism=100, seed=1234, d=2,
    Arrival=random.expovariate, AArgs=0.5,
    Service=random.expovariate, SArgs=1)
sim.RunSim()
```

2.3 Monitors

As simulations run, the *Monitor* class interacts with the main environment, gathering data at certain intervals

This module contains methods for monitoring and visualization. As simulations run, the *Monitor* class interacts with the main environment, gathering data at certain intervals. Moreover, the *Monitor* class was designed to be general enough so that one can build their own by overriding its *Name* and its data-gathering *Add* function.

`class parallelqueue.monitors.`**`JobTime`**

Tracks time of job entry and exit.

`class parallelqueue.monitors.`**`JobTotal`**

Tracks total time each job/set spends in system. To get the mean time each job/set spends:

Example

```
from monitors import JobTotal
import pandas as pd
sim = base_models.RedundancyQueueSystem(maxTime=100.0, parallelism=10, seed=1234,␣
↪d=2, Arrival=random.expovariate,
                              AArgs=0.5, Service=random.expovariate, SArgs=0.2,␣
↪doPrint=True, Monitors = [JobTotal])
sim.RunSim()
totals = sim.MonitorOutput["JobTotal"]
mean = pd.Series(totals, index = totals.keys()).mean()
```

`class parallelqueue.monitors.`**`Monitor`**

Base class defining the behaviour of monitors over *ParallelQueue* models. Unless overridden, the return of this class will be a *dict* of values.

> **Note:** In general, if you need data not provided by any one of the default implementations, you would fare better by overriding elements of *Monitor* as needed. This is as opposed to calling a collection of monitors which will then need to update frequently.

`class parallelqueue.monitors.`**`ReplicaSets`**

Tracks replica sets generated over time, along with their times of creation and disposal.

`class parallelqueue.monitors.`**`TimeQueueSize`**

Tracks queue sizes over time.

CHAPTER THREE

EXAMPLES

3.1 Join the Shortest Queue

This example is a translation of this example from SimPy Classic. In this scenario, we are implementing what is known as the Join the Shortest Queue (JSQ) algorithm, wherein a job will choose the queue with the smallest wait time. In our DES universe, the jobs will be “Customers” and the servers will be “Counters” at a bank. There are two counters currently servicing patrons and the patrons are smart enough to not to join the bigger of the two lines.

To start, let us run our system until either 30 customers are generated and complete their time in the environment or if some period of time (400 arbitrary units of time) have passed. Let us also choose an average time for each customer to spend in the bank and the mean of our arrival process. For simplicity, we will assume both the service and arrival processes to be exponential.

```
from parallelqueue.base_models import JSQd
from parallelqueue.monitors import TimeQueueSize
from random import expovariate


sim = JSQd(parallelism=2, seed=123, d=1, Arrival=expovariate, AArgs=0.10,
           Service=expovariate, SArgs=0.12, maxTime=400,
           numberJobs=30, Modules=[TimeQueueSize])
# Note that because numberJobs is conditional on infiniteJobs being false,
# we manually specify so before running the simulation.
sim.RunSim()
```

Now, with our dataframe, we can visualize the queue loads over time by running:

```
from matplotlib import pyplot as plt
sim.DataFrame.plot()
plt.show()
```

This package aims to allow for easier implementation of novel parallel processing approaches in Python DES packages (especially SimPy).

CHAPTER FOUR

INSTALLATION

From PyPi:

`pip install parallelqueue`

From git repository:

```
git clone https://github.com/aarjaneiro/ParallelQueue
cd ParallelQueue
python setup.py install
```

CHAPTER

FIVE

CURRENT GOALS

1. Introduce more common models into *base_models*.
2. Optimize SimPy boilerplate common to all models by incorporating Cython.
3. Incorporate https://github.com/tqdm/tqdm for better progress visualization and simulation parallelization.

CHAPTER

SIX

INTERESTED IN CONTRIBUTING?

Do feel free to write an issue or submit a PR! If you are interested co-maintaining this package with me, please email me at ajstone@uwaterloo.ca (merely include a brief description of your familiarity with Python and Queueing Theory).

CHAPTER SEVEN

REFERENCES

Heavy Traffic Analysis of the Mean Response Time forLoad Balancing Policies in the Mean Field Regime Tim Hellemans, Benny Van Houdt (2020)

https://arxiv.org/abs/2004.00876

Redundancy-d:The Power of d Choices for Redundancy Kristen Gardner, Mor Harchol-Balter, Alan Scheller-Wolf, Mark Velednitsky, Samuel Zbarsky (2020)

https://doi.org/10.1287/opre.2016.1582

7.1 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX