January 4, 2026

Contents

1 Introduction

To achieve high performance, the simulation logic is: This document describes a high-performance tool designed for the Multiserver-Job Queueing Model (MJQM) [1, 2]. The tool uses the Sub-Perfect Sample algorithm presented in [3] to find the stationary distribution of the system's workload.

To achieve high performance, the simulation logic is:

- GPU-Accelerated: core numerical operations are vectorized and executed on the GPU using the `cupy` library.
- Batch-Processed: the simulation runs thousands of independent trials (`Ntries`) simultaneously in large batches to maximize GPU utilization.
- Modular: the architecture is designed to easily swap out or add new service time distributions, which is the primary focus of this documentation.

The core of the simulation is a backward-simulation approach based on a Lindley-like recursion, which is implemented in the `one_sampling` function. The perfect sampling is achieved by a "doubling" scheme (starting at length `L` and doubling to `2L`, `4L`, ...) until the initial workload state no longer affects the final workload state.

2 System Architecture

The tool is built on three main Python files.

`main.py` This is the main entry point. It parses command-line arguments and acts as a controller. It selects the correct service time distribution based on the `-dist` argument. It selects the appropriate parsing function accordingly, and the corresponding service time generator function and passes them to the simulation engine.

`perfsampl.py` This file is the core of the whole program, and it is the distribution-agnostic simulation engine.

- `perfect_sampling()`: manages the overall simulation, including batching, the doubling-convergence loop, and writing results to the CSV file.
- `generate_chunk()`: generates blocks of random numbers on the CPU (for robust seeding) and then calls the specific `compute_sigmas` function (passed in from `main.py`) to convert these uniform randoms into service times on the GPU.
- `one_sampling()`: performs one vectorized step of the backward simulation. It is completely general and only requires the final `sigmas`, `taus`, and `alphas`.

`distros.py` This file is a library for the different possible distributions. For each supported probability distribution, it must provide two functions:

1. A **parser function** (e.g., `exp_parser`) that knows how to read a line from an input text file and return a dictionary of parameters.
2. A **sigma function** (e.g., `compute_sigmas_exp`) that takes uniform random numbers and generates service times on the GPU according to that distribution.

3 Usage

3.1 Running the Simulation

The simulation is launched from `main.py`. A `run.sh` script is provided to automate the process of running experiments for different distributions and arrival rates (`lambda`).

```
#!/bin/bash

# Fixed Parameters
nsamples_values=(100000)
N=20
L=100
split_size=100
batch_size=10000
device="cuda:0"
seed=42

# Define Distributions
declare -A dist_input
declare -A dist_lambdas

# Exponential
dist_input["exp"]="synth_20_exp"
dist_lambdas["exp"]="1.3338 1.5244 1.7149"

# Erlang-k
dist_input["erlk"]="synth_20_erlk"
dist_lambdas["erlk"]="1.2515 1.4303 1.6091 1.7832"

# Loop Over Distributions
for dist in "${!dist_input[@]}"; do
    in_fname="${dist_input[$dist]}"
    IFS=' ' read -r -a lambdas <<< "${dist_lambdas[$dist]}"

    for lambda in "${lambdas[@]}"; do
        for nsamples in "${nsamples_values[@]}"; do
            echo ">>> Running with lam = $lambda"
            time python3 main.py \
                -dist "$dist" \
                -device "$device" \
                -in_fname "$in_fname" \
                -N "$N" \
                -lam "$lambda" \
                -nsamples "$nsamples" \
                -L "$L" \
                -split_size "$split_size" \
                -batch_size "$batch_size" \
                -seed "$seed"
        done
    done
done
```

Listing 1: Example `run.sh` script

3.2 Command-Line Arguments

The `main.py` script accepts the following arguments.

Table 1: Command-Line Arguments

Argument	Type	Description
`-dist`	str	**Required.** The distribution to use. Choices: `exp`, `bpar`, `erlk`, `hyperexp`.
`-device`	str	The GPU device to use (e.g., `cuda:0`).
`-in_fname`	str	Name of the input parameter file (e.g., `synth_20_exp`) located in the `inputs/` folder.
`-N`	int	Number of servers in the system.
`-lam`	float	The overall arrival rate (lambda).
`-nsamples`	int	Total number of simulation trials to run.
`-L`	int	The *initial* simulation length for the doubling algorithm.
`-split_size`	int	The chunk size (L) for `one_sampling`. A large `-L` is split into multiple chunks of this size to avoid GPU OOM errors.
`-batch_size`	int	The number of trials to run in a single batch. `-nsamples` is split into batches of this size to avoid CPU/GPU OOM errors.
`-seed`	int	The master seed for the random number generator to provider reproducibility.

3.3 Input and Output

- **Inputs:** All parameter input files (e.g., `synth_20_exp.txt`) are expected to be in a directory named `inputs/`.
- **Outputs:** All simulation results are saved as `.csv` files in a directory named `results/`.

4 How to Add a New Distribution

This is the main purpose of the tool's design. To add a new service time distribution (e.g., the Log-Normal distribution), you must complete four steps.

4.1 Step 1: Define the Input File Format

First, decide what parameters your distribution needs for each class. Create a `.txt` file in the `inputs/` directory.

For our Log-Normal example, let's say we need *mean* and *variance* of the underlying Normal distribution. The format might be: `(size, prob, mu, sigma_sq)`. A file `inputs/synth_20_lognormal.txt` might look like:

```
(5, 0.5, 0.1, 0.2)
(10, 0.5, 0.5, 0.3)
```

4.2 Step 2: Create a Parser Function in `distros.py`

Add a new parser function to `distros.py` that can read one line of your new file. It must return a dictionary.

```python
def lognormal_parser(values):
    """
    Parses a line for Log-Normal distribution input parameters.
    Expected format: (size, prob, mu, sigma_sq)
    """
    return {
        'sizes': int(values[0]),
        'probs': float(values[1]),
        'mus': float(values[2]),
        'sigma_sqs': float(values[3])
    }
```

Listing 2: New parser function in `distros.py`

4.3 Step 3: Implement the Sigma Function in `distros.py`

This is the most important step. You must write a GPU-accelerated function that generates the service times.

Function Signature: The function **must** have the signature:

`def compute_sigmas_lognormal(rng_sigma, class_indices, **params):`

- `rng_sigma`: This is a **list of CuPy arrays** containing uniform random numbers between [0, 1). The number of arrays in the list is determined by `num_streams` set in `main.py`.
- `class_indices`: This is a 2D CuPy array (`B, L`) that maps each arrival to its class ID.
- `**params`: This is a dictionary containing the parameter arrays loaded by your parser (e.g., `params['mus']`, `params['sigma_sqs']`).

Example Implementation: To generate a Log-Normal variate, we can use two uniform randoms (`U1`, `U2`) with the Box-Muller transform to get a standard normal variate `Z`, and then transform it. This means we will need **2 random number streams**.

```python
import cupy as cp

def compute_sigmas_lognormal(rng_sigma, class_indices, **params):
    """
    Generates Log-Normal distributed service times (sigmas) on the GPU.
    Uses the Box-Muller transform, requiring 2 random streams.
    """

    # --- 1. Get Random Streams ---
    # We need two independent uniform random streams
    unif_1 = rng_sigma[0]
    unif_2 = rng_sigma[1]

    # --- 2. Get Class-Specific Parameters ---
    # Use class_indices to "map" arrivals to their parameters
    mus = params['mus'][class_indices]
    sigma_sqs = params['sigma_sqs'][class_indices]
    sigmas_norm = cp.sqrt(sigma_sqs) # StDev of the normal

    # --- 3. Implement the Distribution (Inverse Transform, etc.) ---
    # Box-Muller transform to get a standard normal variate (Z)
    # We only need one of the two generated variates.
    Z = cp.sqrt(-2.0 * cp.log(unif_1)) * cp.cos(2.0 * cp.pi * unif_2)

    # --- 4. Scale and Shift ---
    # Transform the standard normal (Z) to N(mu, sigma_sq)
    log_Y = mus + Z * sigmas_norm

    # Exponentiate to get the Log-Normal variate
    sigmas_final = cp.exp(log_Y)

    return sigmas_final
```

Listing 3: New sigma function in `distros.py`

4.4 Step 4: Register in `main.py`

Finally, add an `elif` block to `main.py` to tell the controller about your new "lognormal" strategy.

```python
# ... other elif blocks ...
elif args.dist == 'erlk':
    parser = distros.erlk_parser
    in_data, nClasses = distros.load_params(args.in_fname, parser)
    num_streams = int(cp.max(in_data['ks']))
    compute_sigmas_foo = distros.compute_sigmas_erlk
    distro_label = 'erlk'

# --- ADD YOUR NEW BLOCK HERE ---
elif args.dist == 'lognormal':
    parser = distros.lognormal_parser # <-- Your new parser
    in_data, nClasses = distros.load_params(args.in_fname, parser)
    num_streams = 2                   # <-- We need 2 for Box-Muller
    compute_sigmas_foo = distros.compute_sigmas_lognormal # <-- Your
   new func
    distro_label = 'lognormal'
# --- END OF NEW BLOCK ---

else:
```

```
19      print(f"Error: Unknown distribution '{args.dist}'")
```

Listing 4: Modification to `main.py`

After these four steps, you can run the simulation by passing the new argument:

```
python3 main.py -dist "lognormal" -in_fname "synth_20_lognormal" ...
```

References

[1] M. Harchol-Balter, "The Multiserver Job Queueing Model," *Queueing Systems: Theory and Applications*, vol. 100, pp. 201–203, 2022.

[2] D. Olliaro, M. Ajmone Marsan, S. Balsamo, and A. Marin, "The saturated Multiserver Job Queuing Model with two classes of jobs: Exact and approximate results," *Performance Evaluation*, vol. 162, p. 102370, 2023.

[3] F. Baccelli, D. Olliaro, M. Ajmone Marsan, and A. Marin, "The Multiserver-Job Stochastic Recurrence Equation for Cloud Computing Performance Evaluation," *pomacs wannabe*, hopefully 2026.