Splitwiser

Efficient LLM inference with Constrained Resources

Computer Systems and Machine Learning

Asad Aali, Adney Cardoza, Melissa Capo

Electrical and Computer Engineering

TEXAS

The University of Texas at Austin

Goal of Project

Receive the benefits of **Split Phase** Inference

using just a **single GPU**

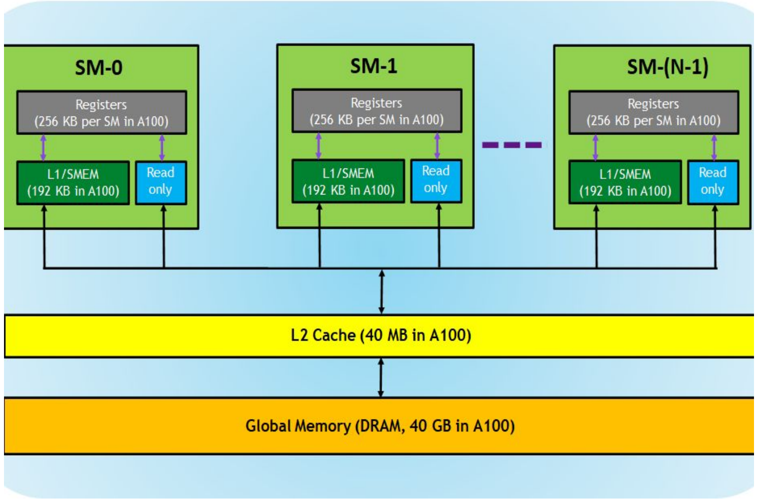
alleviating the **overhead** of **moving data** between multiple GPUs.

Profiling Results

- From our midterm profiling results, it was evident that **splitting phases** enabled a much **lower end to end inference time**.
- Additionally, the programming model for coordinating tasks across devices is fairly specified. However, **fine grained resource management** on single device requires **non-trivial effort**.
- This talk will provide some **GPU specific background** before explaining our solution attempts.

GPU Specific Background

- Parallel Processing Capable Architecture
- Compute Execution Procedure
  - Copy required input data onto device memory
  - Execute compute defined by a CUDA kernels
  - Copy back output to host memory



GPU Specific Background

- Relevant Metrics
  - Using profiling tool: nvtx - sm_throughput, dram_throughput
- Multiprocessing/MPS
  - Multiprocessing - perspective of CPU scheduling of compute related to task.
  - MPS - spawns one server, and multiple kernels are wrapped inside MPS client, hence GPU resources can be throttled as required.
- Device Memory Optimization
  - All processes share the same model - no duplication → space savings

Solution Attempt 1: Sending data between processes

- Big overhead - 8GB file
- Lots of synchronization overhead, lesser performance.
- Interprocess communication for big objects is not possible without involving host -> big disadvantage

Solution Attempt 2: Coarse Grained Scheduler

A. Hugging Face Pipeline + MP/MPS
B. vLLM + MP/MPS

Splitwiser Inference with Hugging Face

- Model: OPT-125
- Dataset: Radiology (CT and MR) reports from MIMIC-III
  - De-identified and publicly-available collection of medical records
  - 30,000 pre-processed inputs
- Max Input Tokens: 512
- Max Output Tokens: 20
- GPU:
  - A10
  - A100
- Batch Size: 20

Source: https://physionet.org/content/mimiciii/1.4/

Comparing Hugging Face with Splitwise Implementation

Feature	Hugging Face Transformers	Splitwise Paper
Separation Level	Partial	Full
Functionality	**Preprocess prompt**, use encoded input in token gen	All tokens in input prompt run through the forward pass of the model to generate the **first output token**
Hardware Utilization	Potentially move some processing off main GPU	Utilize GPU

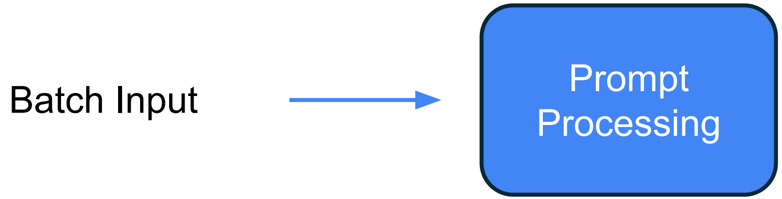
Sequential (1 Process)

```
# generate summary for each finding
t0 = time.time()
torch.cuda.nvtx.range_push(f'token0')
for step, batch in tqdm(enumerate(test_loader)):
    batch = {k: v.to(device) for k, v in batch.items()}
    with torch.no_grad():
        outputs = model.generate(input_ids=batch['input_ids'], attention_mask=batch['attention_mask']
torch.cuda.nvtx.range_pop()
```

Sequential (1 Process)

```
# generate summary for each finding
t0 = time.time()
torch.cuda.nvtx.range_push(f'token0')
for step, batch in tqdm(enumerate(test_loader)):
    batch = {k: v.to(device) for k, v in batch.items()}
    with torch.no_grad():
        outputs = model.generate(input_ids=batch['input_ids'], attention_mask=batch['attention_mask']
torch.cuda.nvtx.range_pop()
```

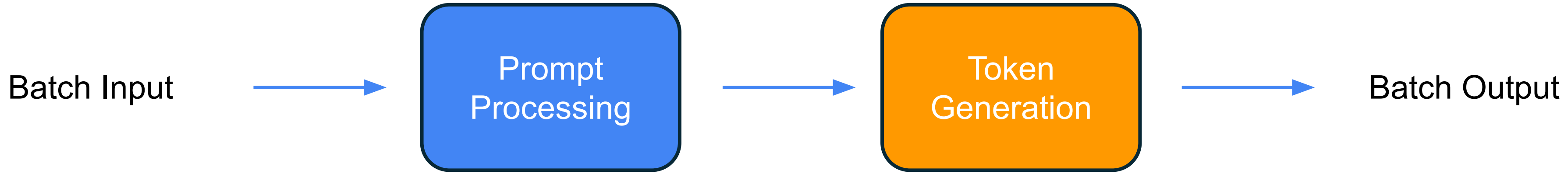
Phases



Sequential (1 Process)

```
# generate summary for each finding
t0 = time.time()
torch.cuda.nvtx.range_push(f'token0')
for step, batch in tqdm(enumerate(test_loader)):
    batch = {k: v.to(device) for k, v in batch.items()}
    with torch.no_grad():
        outputs = model.generate(input_ids=batch['input_ids'], attention_mask=batch['attention_mask']
torch.cuda.nvtx.range_pop()
```

Phases



Sequential (1 Process)

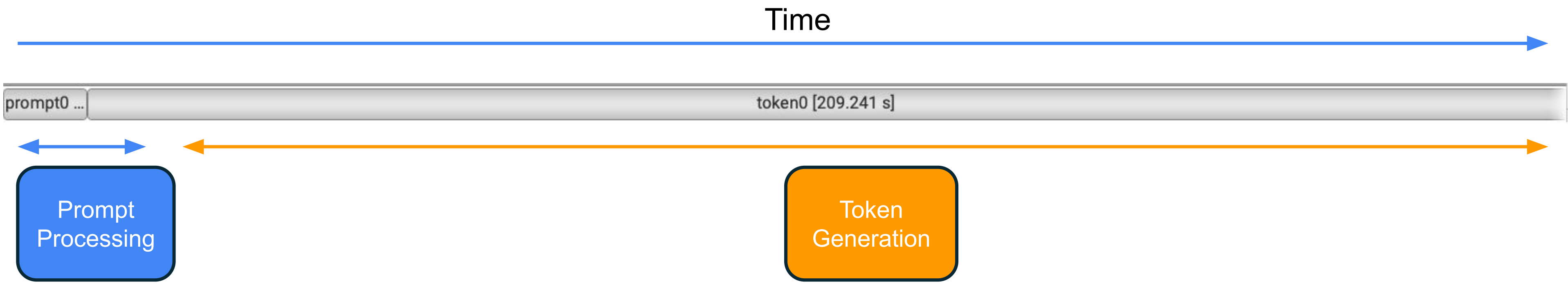
```python
# generate summary for each finding
t0 = time.time()
torch.cuda.nvtx.range_push(f'token0')
for step, batch in tqdm(enumerate(test_loader)):
    batch = {k: v.to(device) for k, v in batch.items()}
    with torch.no_grad():
        outputs = model.generate(input_ids=batch['input_ids'], attention_mask=batch['attention_mask']
torch.cuda.nvtx.range_pop()
```

Nsight Output

Sequential (1 Process)

```python
# generate summary for each finding
t0 = time.time()
torch.cuda.nvtx.range_push(f'token0')
for step, batch in tqdm(enumerate(test_loader)):
    batch = {k: v.to(device) for k, v in batch.items()}
    with torch.no_grad():
        outputs = model.generate(input_ids=batch['input_ids'], attention_mask=batch['attention_mask']
torch.cuda.nvtx.range_pop()
```

Nsight Output

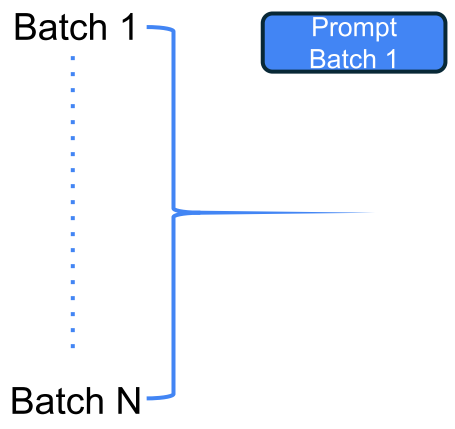


Splitwiser Design

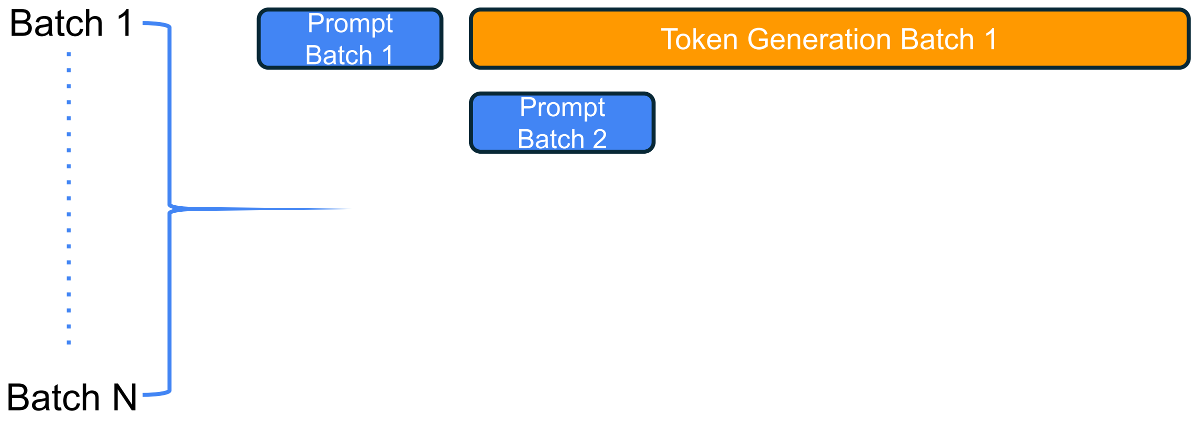
Batch 1

Batch N

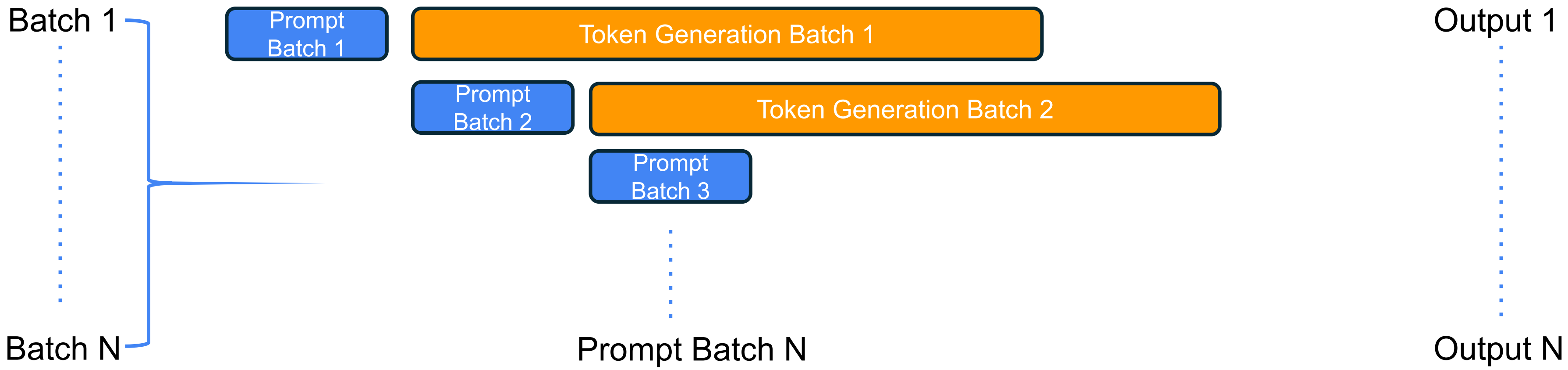
Splitwiser Design



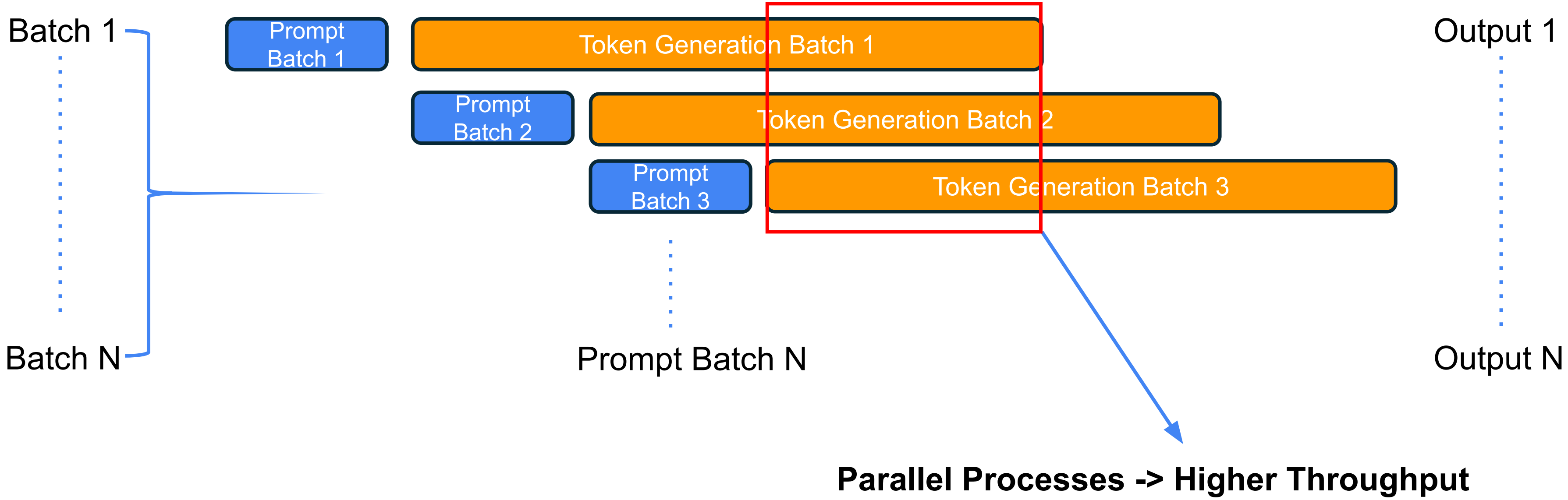
Splitwiser Design



Splitwiser Design



Splitwiser Design



Splitwiser (Implementation)

```
# generate summary for each finding
t0 = time.time()
token_processes = []
prompt_processes = []
for rank in range(args.nproc):
    # Prompt Phase
    prompt_queue = mp.Queue()
    prompt_proc = mp.Process(target=process.get_loader, args=(datasets[rank], tokenizer
    prompt_proc.start()
    prompt_processes.append(prompt_proc)
    dataloader = prompt_queue.get()

    # Token Phase
    token_proc = mp.Process(target=process_stream, args=(dataloader, model, rank))
    token_proc.start()
    token_processes.append(token_proc)

for prompt_proc, token_proc in zip(prompt_processes, token_processes):
    prompt_proc.join()
    token_proc.join()
```

Splitwiser (Implementation)

```python
# generate summary for each finding
t0 = time.time()
token_processes = []
prompt_processes = []
for rank in range(args.nproc):
    # Prompt Phase
    prompt_queue = mp.Queue()
    prompt_proc = mp.Process(target=process.get_loader, args=(datasets[rank], tokenizer
    prompt_proc.start()
    prompt_processes.append(prompt_proc)
    dataloader = prompt_queue.get()

    # Token Phase
    token_proc = mp.Process(target=process_stream, args=(dataloader, model, rank))
    token_proc.start()
    token_processes.append(token_proc)

for prompt_proc, token_proc in zip(prompt_processes, token_processes):
    prompt_proc.join()
    token_proc.join()
```

Nsight Output

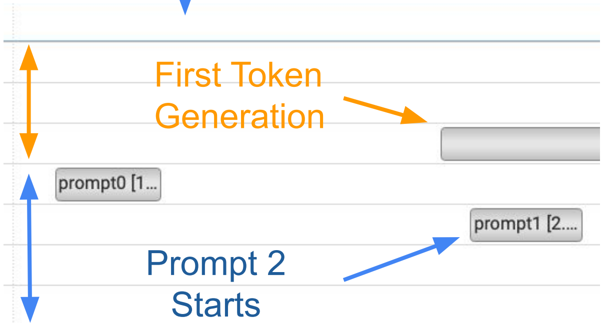
Splitwiser (Implementation)

```python
# generate summary for each finding
t0 = time.time()
token_processes = []
prompt_processes = []
for rank in range(args.nproc):
    # Prompt Phase
    prompt_queue = mp.Queue()
    prompt_proc = mp.Process(target=process.get_loader, args=(datasets[rank], tokenizer
    prompt_proc.start()
    prompt_processes.append(prompt_proc)
    dataloader = prompt_queue.get()

    # Token Phase
    token_proc = mp.Process(target=process_stream, args=(dataloader, model, rank))
    token_proc.start()
    token_processes.append(token_proc)

for prompt_proc, token_proc in zip(prompt_processes, token_processes):
    prompt_proc.join()
    token_proc.join()
```

Nsight Output



Splitwiser (Implementation)

```python
# generate summary for each finding
t0 = time.time()
token_processes = []
prompt_processes = []
for rank in range(args.nproc):
    # Prompt Phase
    prompt_queue = mp.Queue()
    prompt_proc = mp.Process(target=process.get_loader, args=(datasets[rank], tokenizer
    prompt_proc.start()
    prompt_processes.append(prompt_proc)
    dataloader = prompt_queue.get()

    # Token Phase
    token_proc = mp.Process(target=process_stream, args=(dataloader, model, rank))
    token_proc.start()
    token_processes.append(token_proc)

for prompt_proc, token_proc in zip(prompt_processes, token_processes):
    prompt_proc.join()
    token_proc.join()
```

Nsight Output

Splitwiser (Implementation)

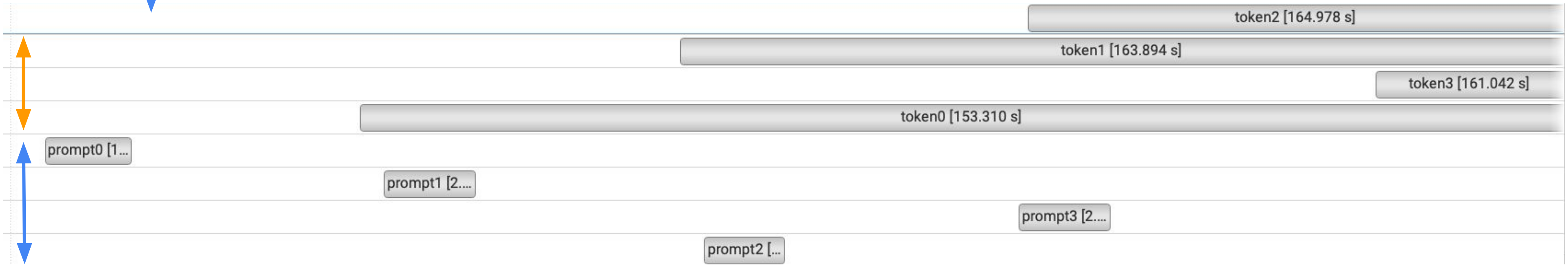
```python
# generate summary for each finding
t0 = time.time()
token_processes = []
prompt_processes = []
for rank in range(args.nproc):
    # Prompt Phase
    prompt_queue = mp.Queue()
    prompt_proc = mp.Process(target=process.get_loader, args=(datasets[rank], tokenizer
    prompt_proc.start()
    prompt_processes.append(prompt_proc)
    dataloader = prompt_queue.get()

    # Token Phase
    token_proc = mp.Process(target=process_stream, args=(dataloader, model, rank))
    token_proc.start()
    token_processes.append(token_proc)

for prompt_proc, token_proc in zip(prompt_processes, token_processes):
    prompt_proc.join()
    token_proc.join()
```

Waiting for all processes to finish

Nsight Output



Splitwiser (Implementation)

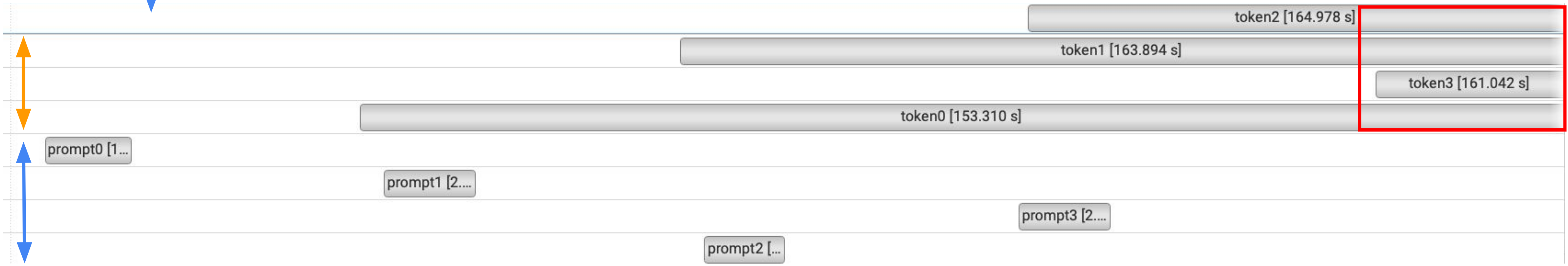
```python
# generate summary for each finding
t0 = time.time()
token_processes = []
prompt_processes = []
for rank in range(args.nproc):
    # Prompt Phase
    prompt_queue = mp.Queue()
    prompt_proc = mp.Process(target=process.get_loader, args=(datasets[rank], tokenizer
    prompt_proc.start()
    prompt_processes.append(prompt_proc)
    dataloader = prompt_queue.get()

    # Token Phase
    token_proc = mp.Process(target=process_stream, args=(dataloader, model, rank))
    token_proc.start()
    token_processes.append(token_proc)

for prompt_proc, token_proc in zip(prompt_processes, token_processes):
    prompt_proc.join()
    token_proc.join()
```

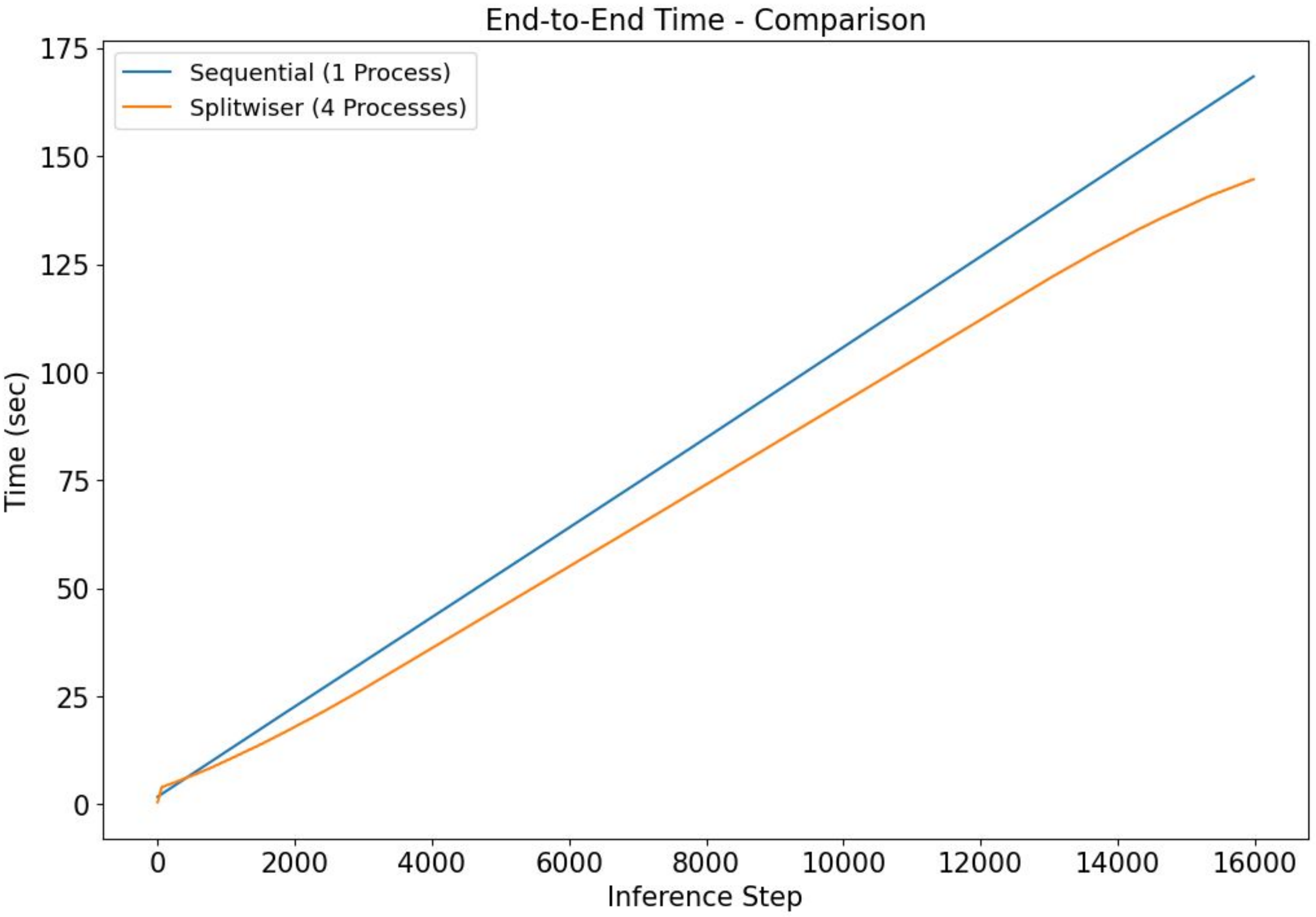
Nsight Output

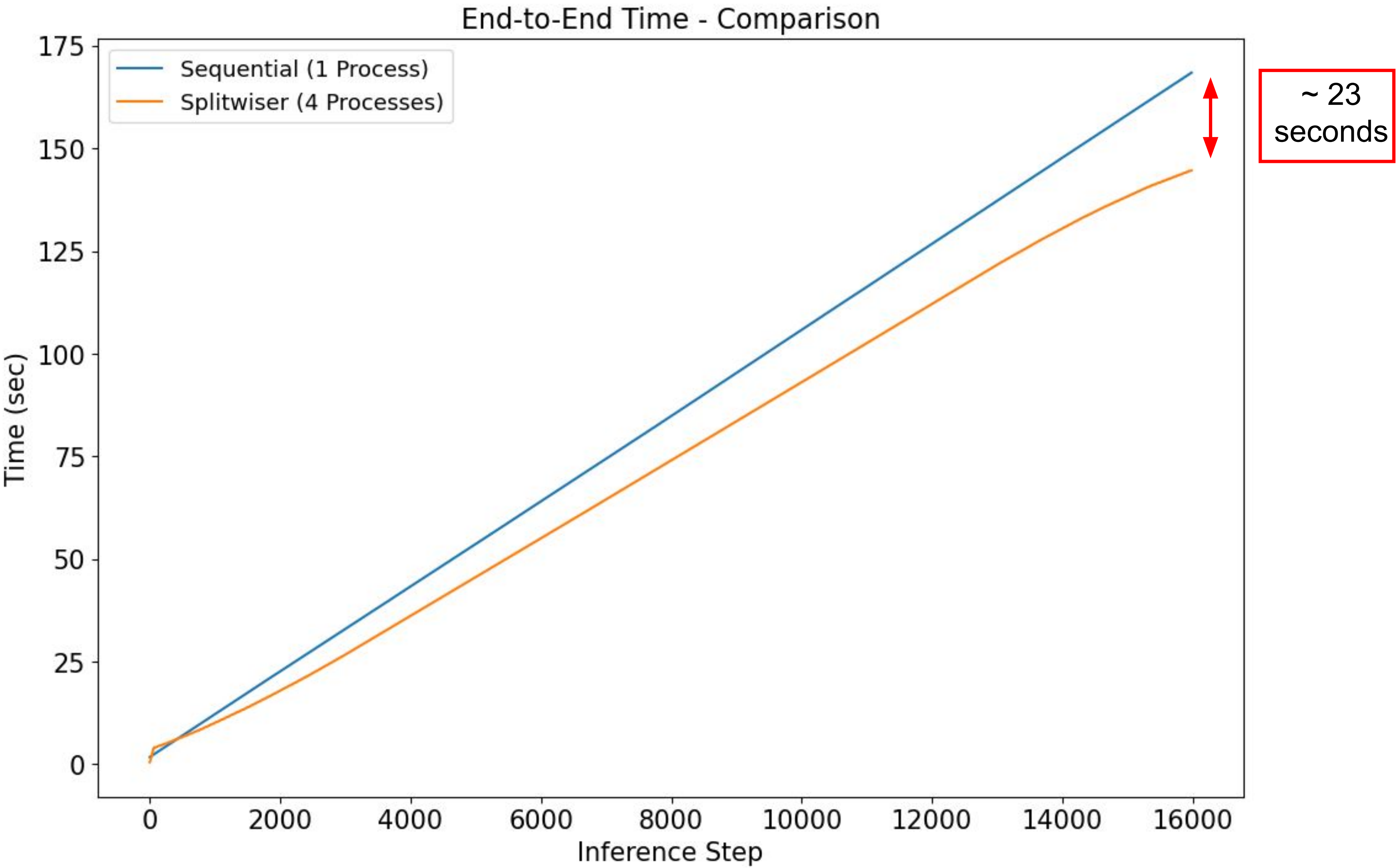
Parallel Token Generation



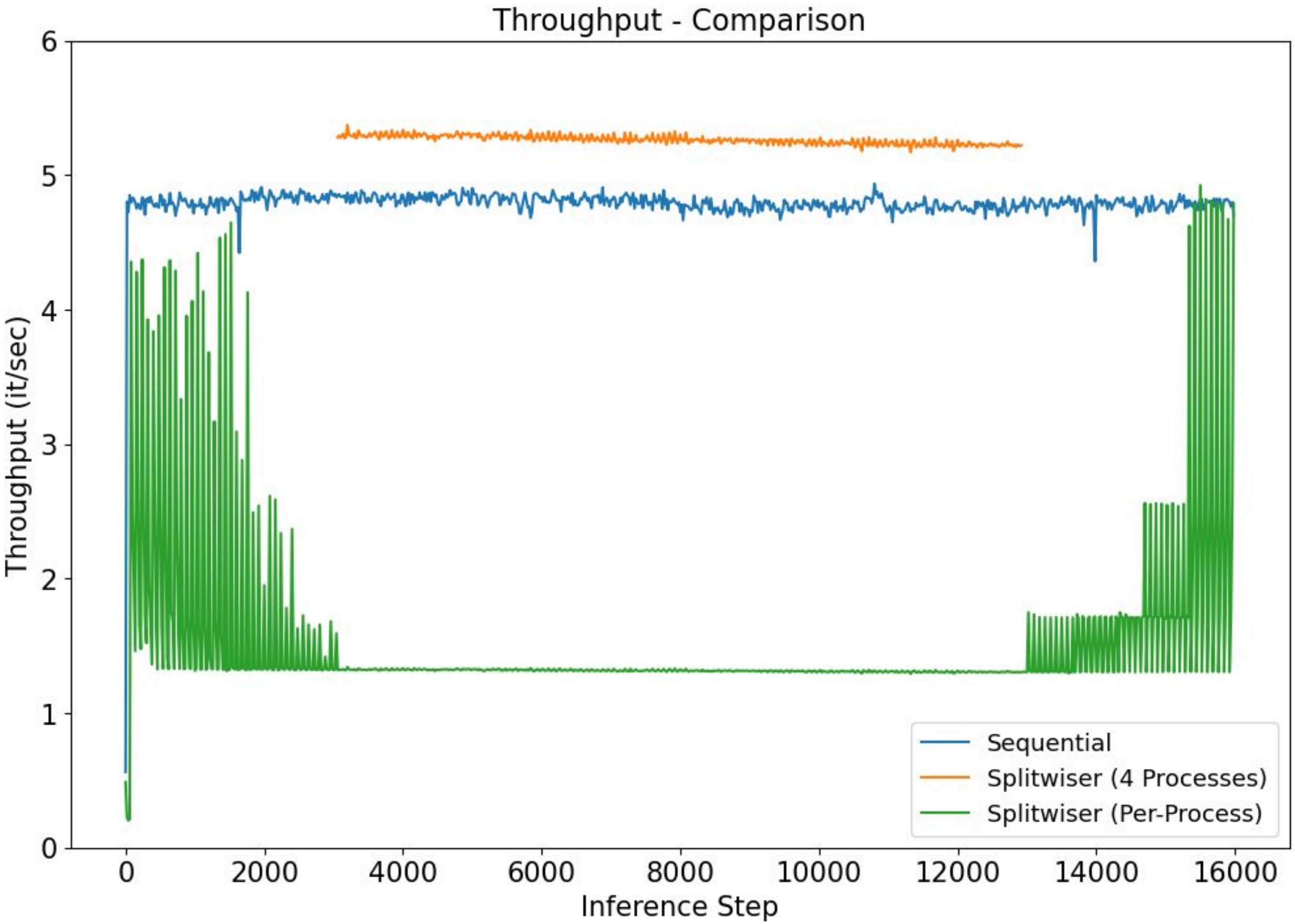
Solution Attempt 2A - Hugging Face Pipeline

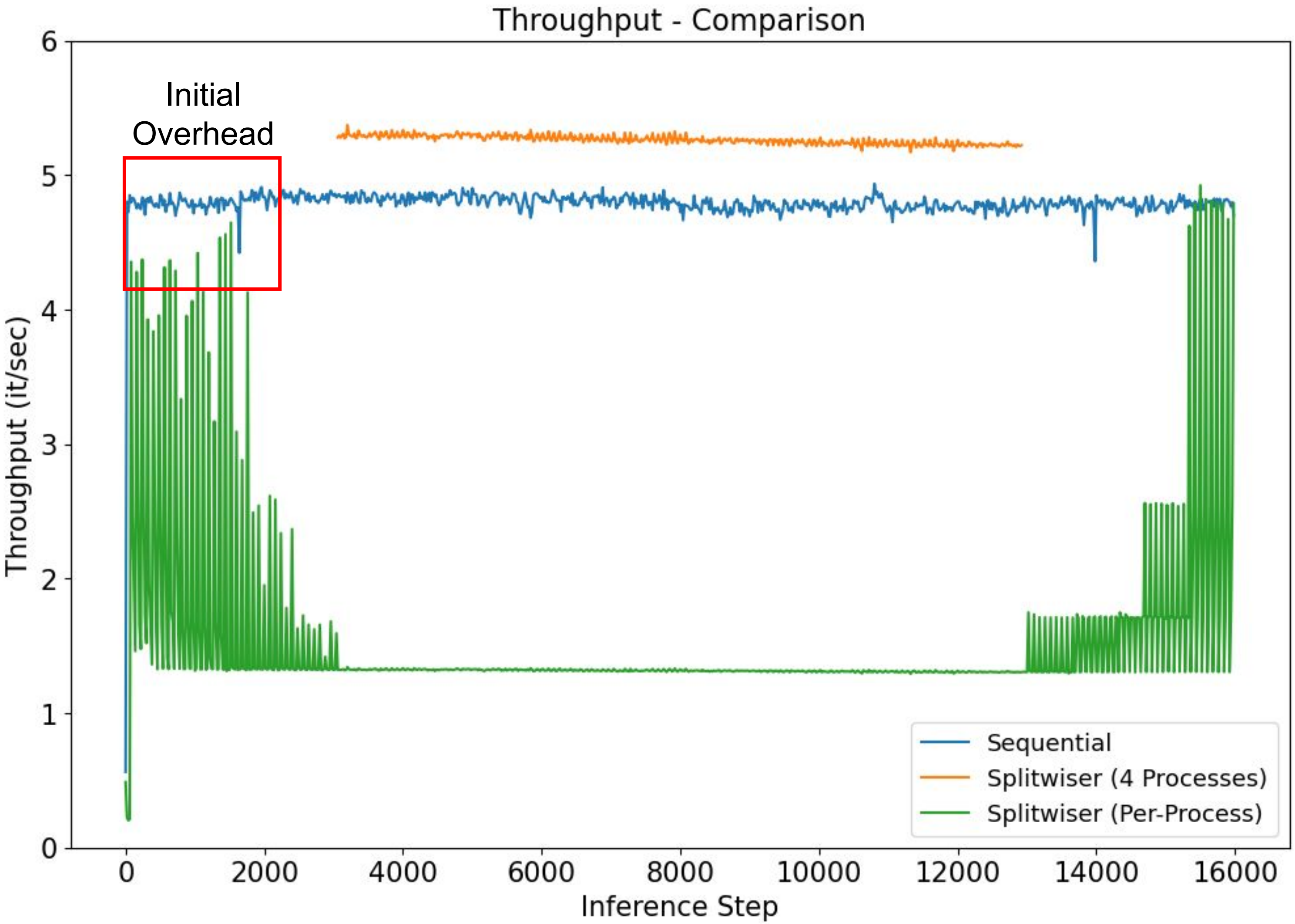
Experiments and Results


End-to-End Time - Comparison
Sequential (1 Process)
Splitwiser (4 Processes)
Time (sec)
Inference Step
0
25
50
75
100
125
150
175
0
2000
4000
6000
8000
10000
12000
14000
16000

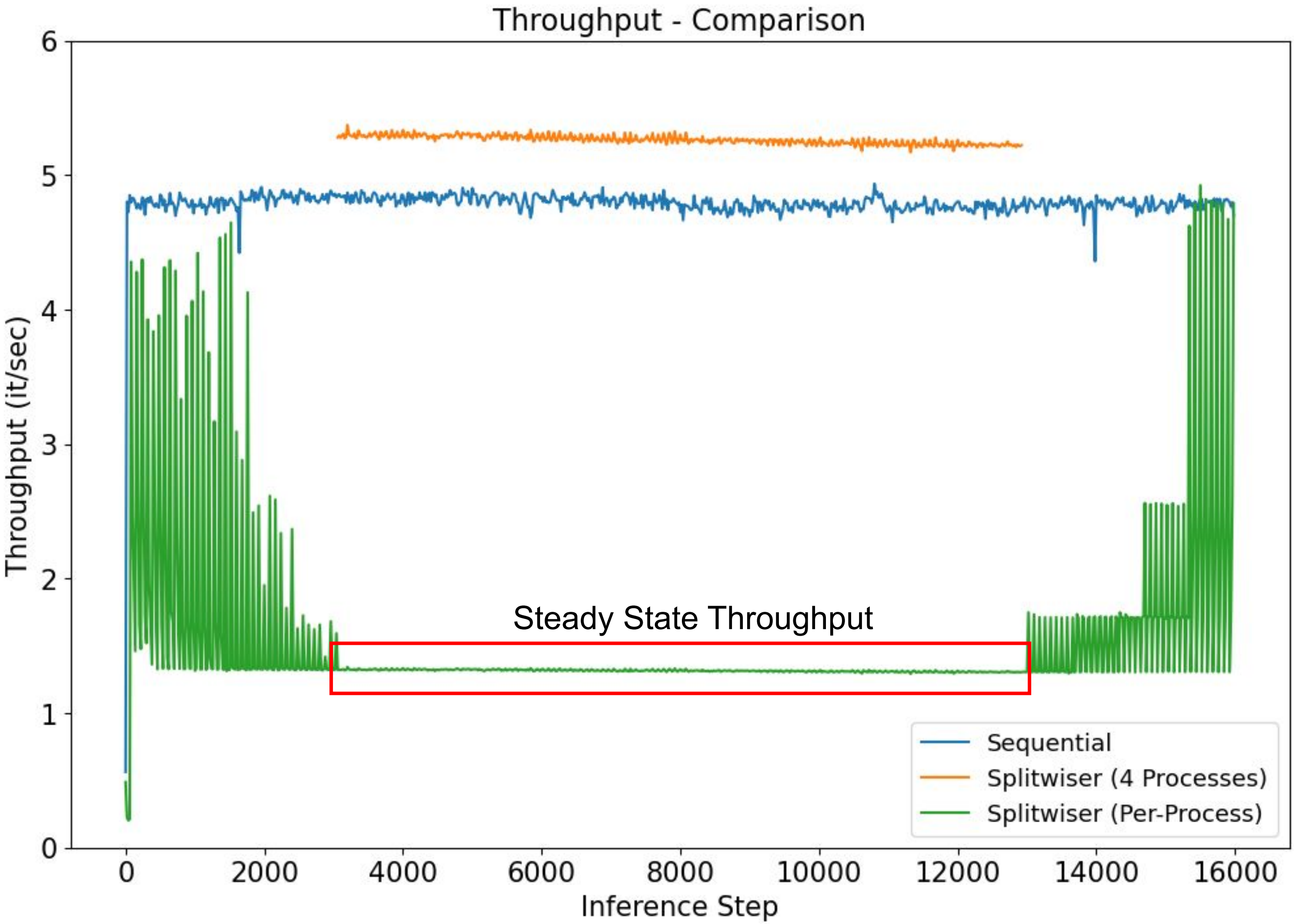


End-to-End Time - Comparison
Sequential (1 Process)
Splitwiser (4 Processes)
Time (sec)
Inference Step
0
25
50
75
100
125
150
175
0
2000
4000
6000
8000
10000
12000
14000
16000
~ 23
seconds

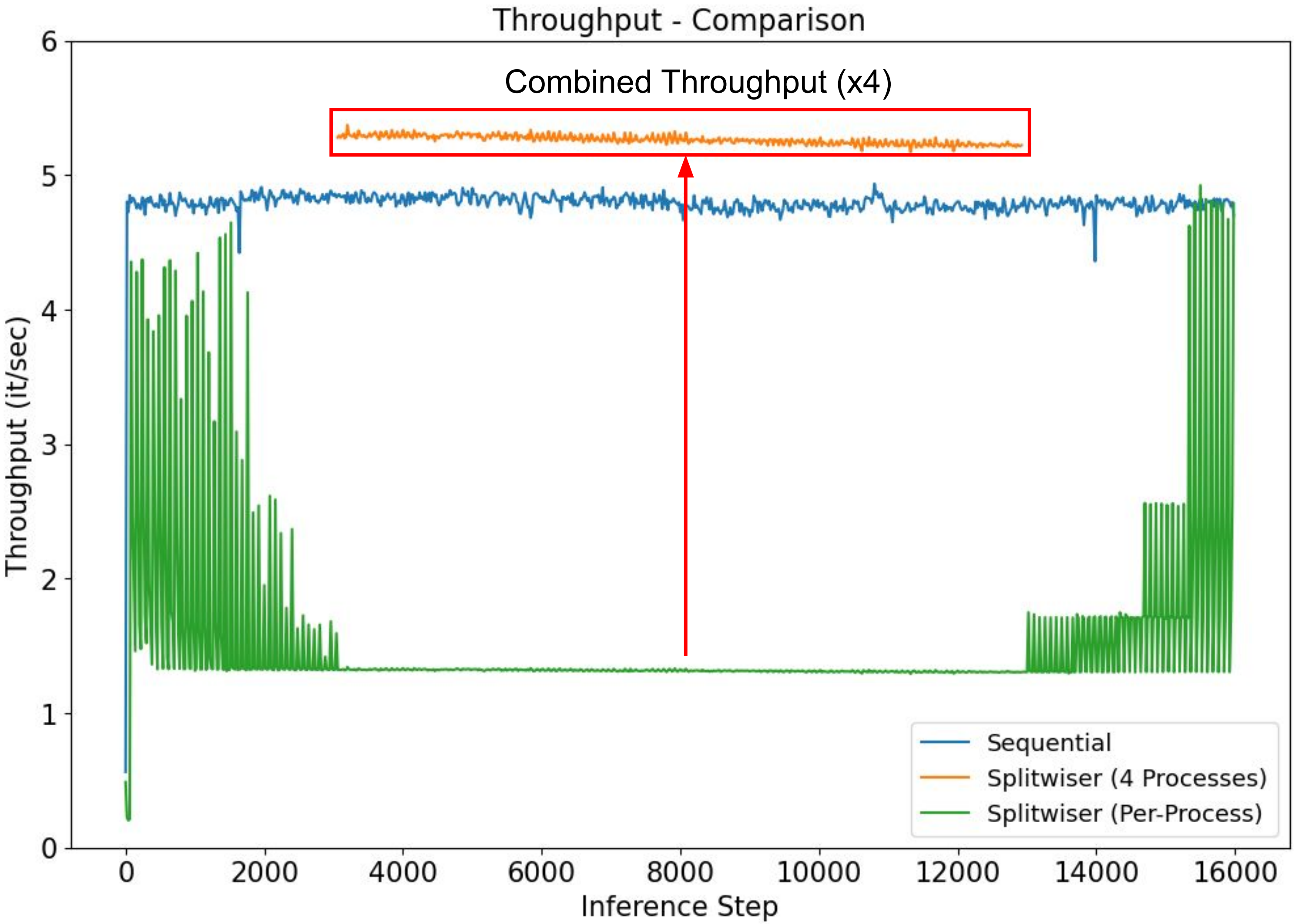

Throughput - Comparison
Throughput (it/sec)
Inference Step
0
1
2
3
4
5
6
0
2000
4000
6000
8000
10000
12000
14000
16000
Sequential
Splitwiser (4 Processes)
Splitwiser (Per-Process)



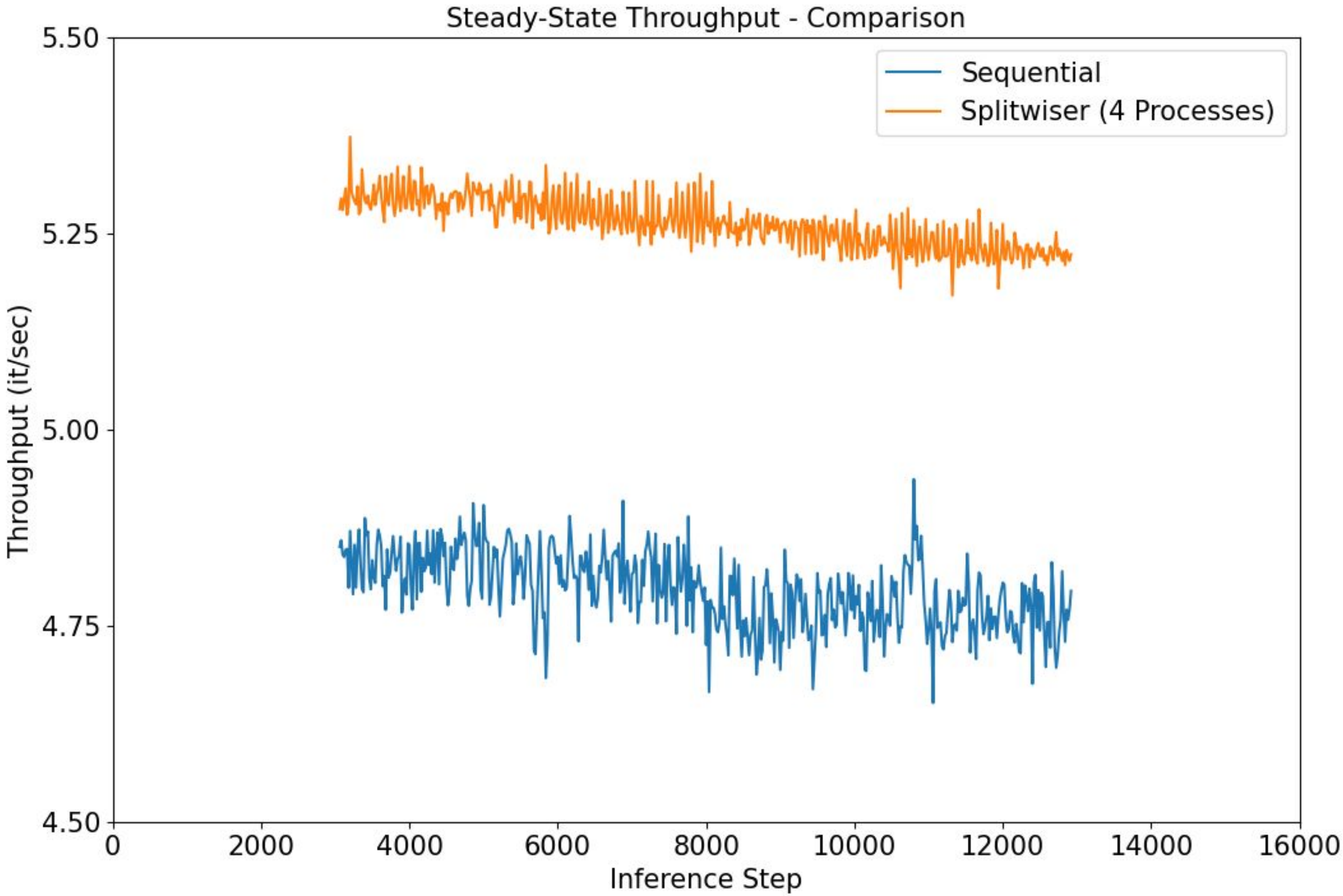
Throughput - Comparison
Initial
Overhead
Throughput (it/sec)
Inference Step
0
1
2
3
4
5
6
0
2000
4000
6000
8000
10000
12000
14000
16000
Sequential
Splitwiser (4 Processes)
Splitwiser (Per-Process)

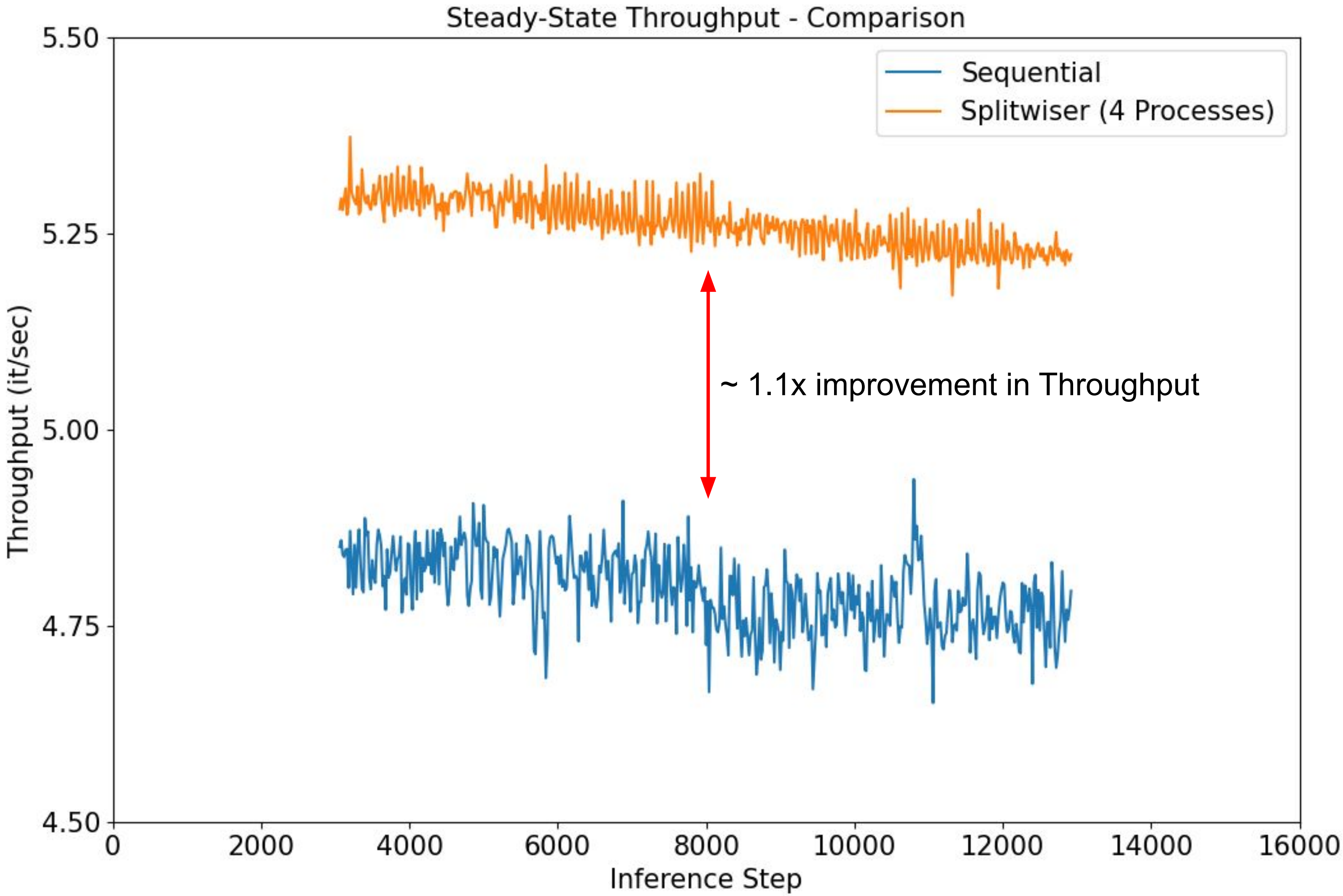


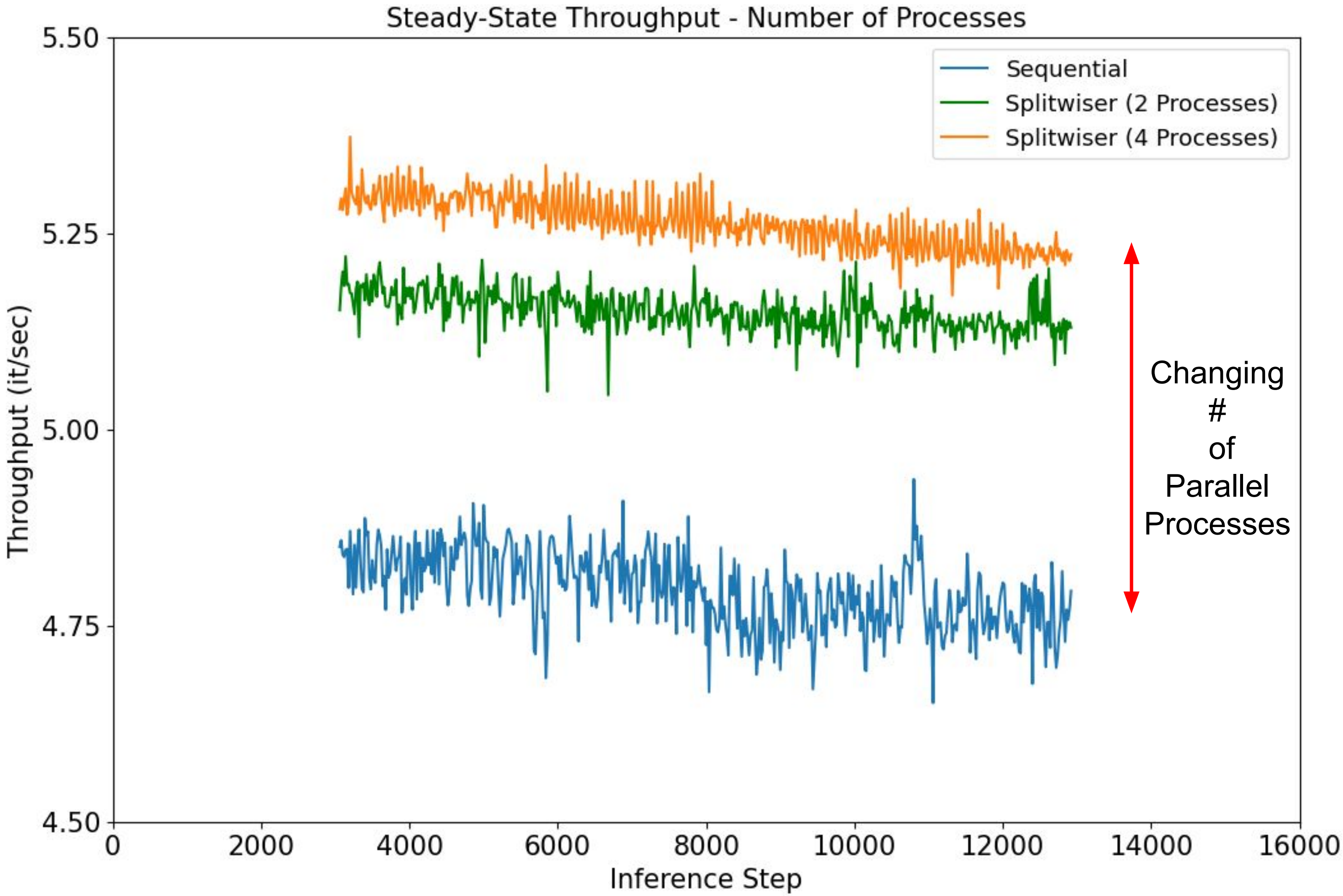
Throughput - Comparison
Steady State Throughput
Throughput (it/sec)
Inference Step
Sequential
Splitwiser (4 Processes)
Splitwiser (Per-Process)
0
1
2
3
4
5
6
0
2000
4000
6000
8000
10000
12000
14000
16000



Throughput - Comparison
Combined Throughput (x4)
Throughput (it/sec)
Inference Step
0
1
2
3
4
5
6
0
2000
4000
6000
8000
10000
12000
14000
16000
Sequential
Splitwiser (4 Processes)
Splitwiser (Per-Process)


Steady-State Throughput - Comparison
Sequential
Splitwiser (4 Processes)
Throughput (it/sec)
Inference Step
5.50
5.25
5.00
4.75
4.50
0
2000
4000
6000
8000
10000
12000
14000
16000


Steady-State Throughput - Comparison
Sequential
Splitwiser (4 Processes)
~ 1.1x improvement in Throughput
Throughput (it/sec)
5.50
5.25
5.00
4.75
4.50
0
2000
4000
6000
8000
10000
12000
14000
16000
Inference Step

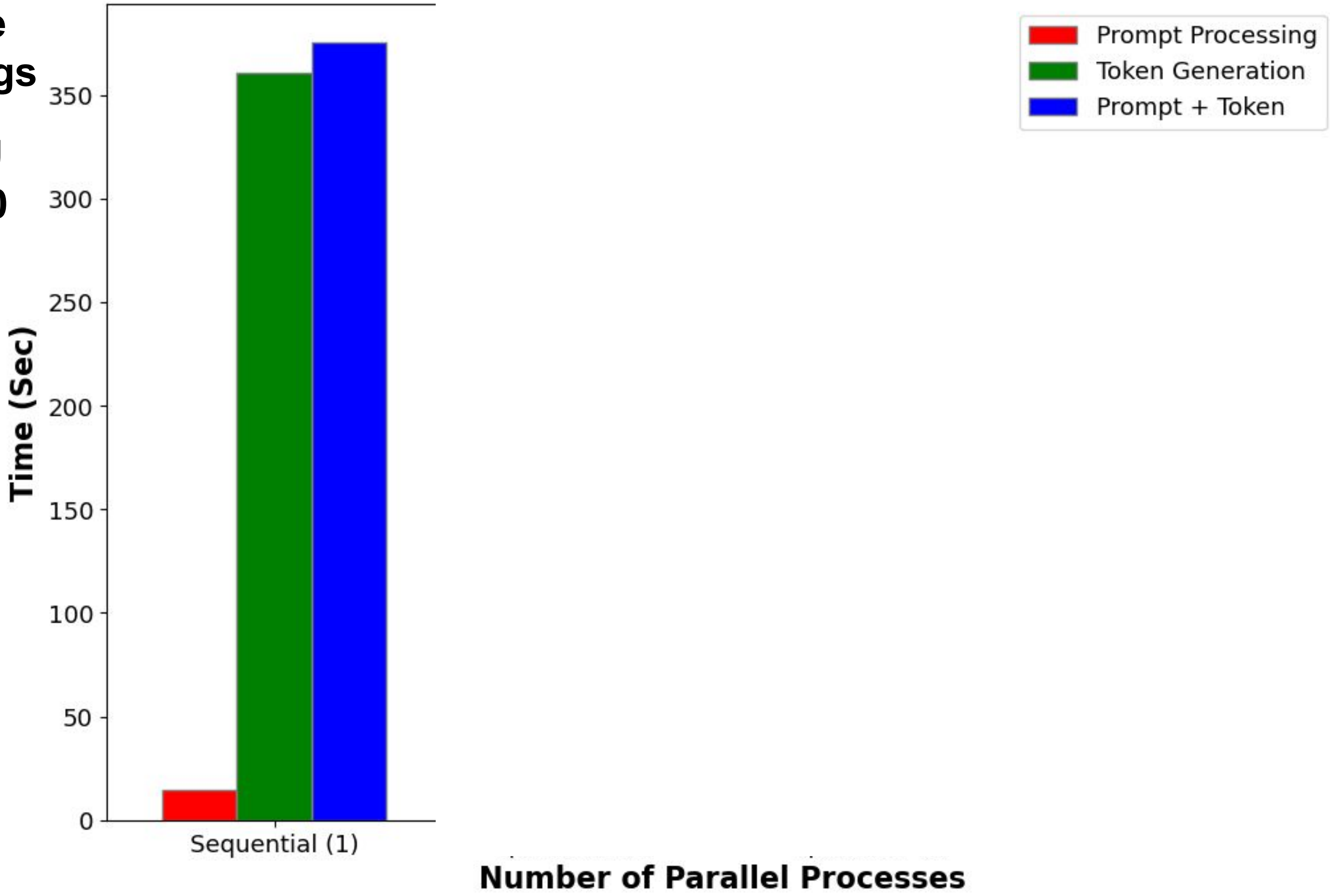

Steady-State Throughput - Number of Processes
Throughput (it/sec)
Inference Step
Sequential
Splitwiser (2 Processes)
Splitwiser (4 Processes)
Changing # of Parallel Processes
5.50
5.25
5.00
4.75
4.50
0
2000
4000
6000
8000
10000
12000
14000
16000

Time Savings

GPU A100

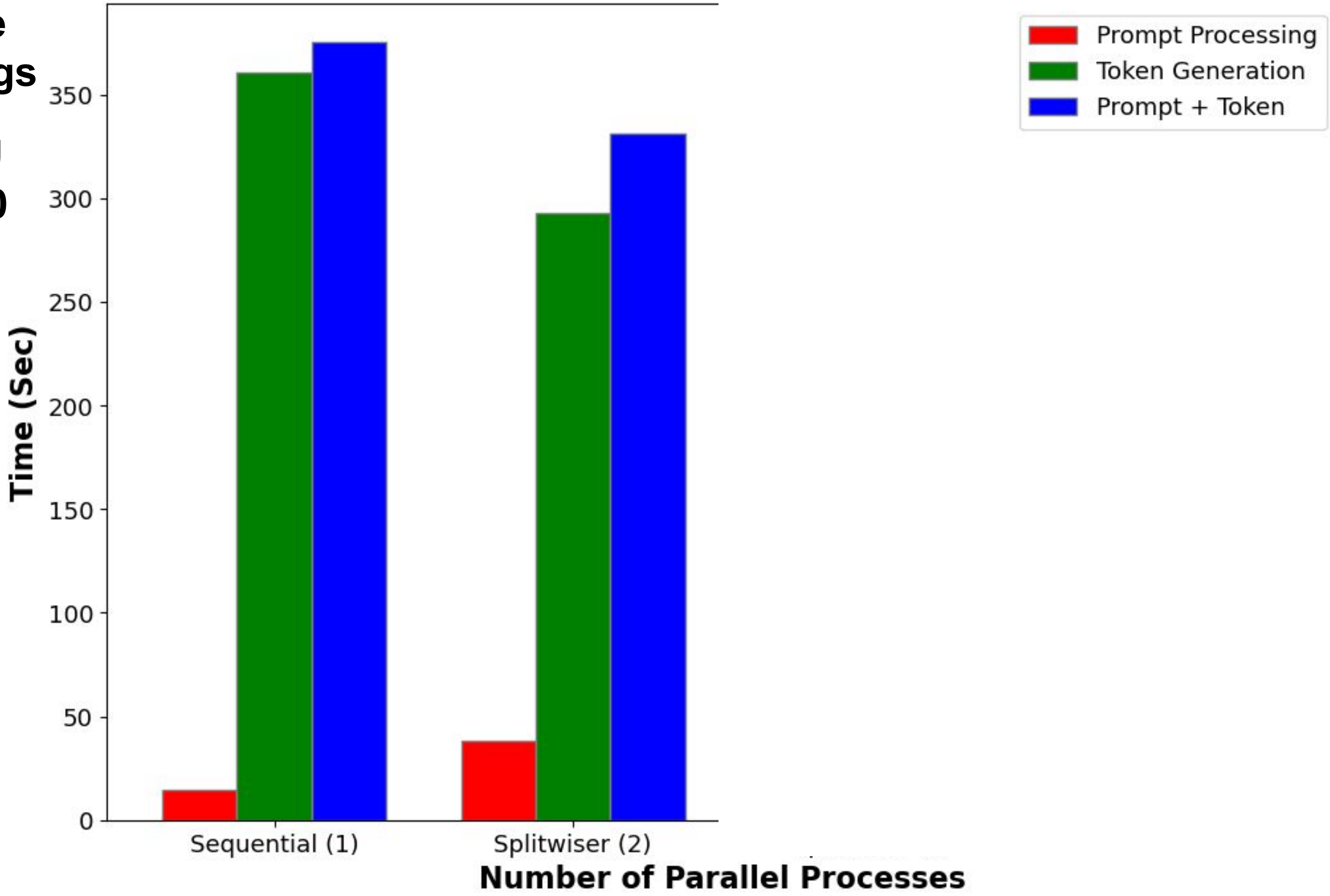
Time Savings

GPU A100



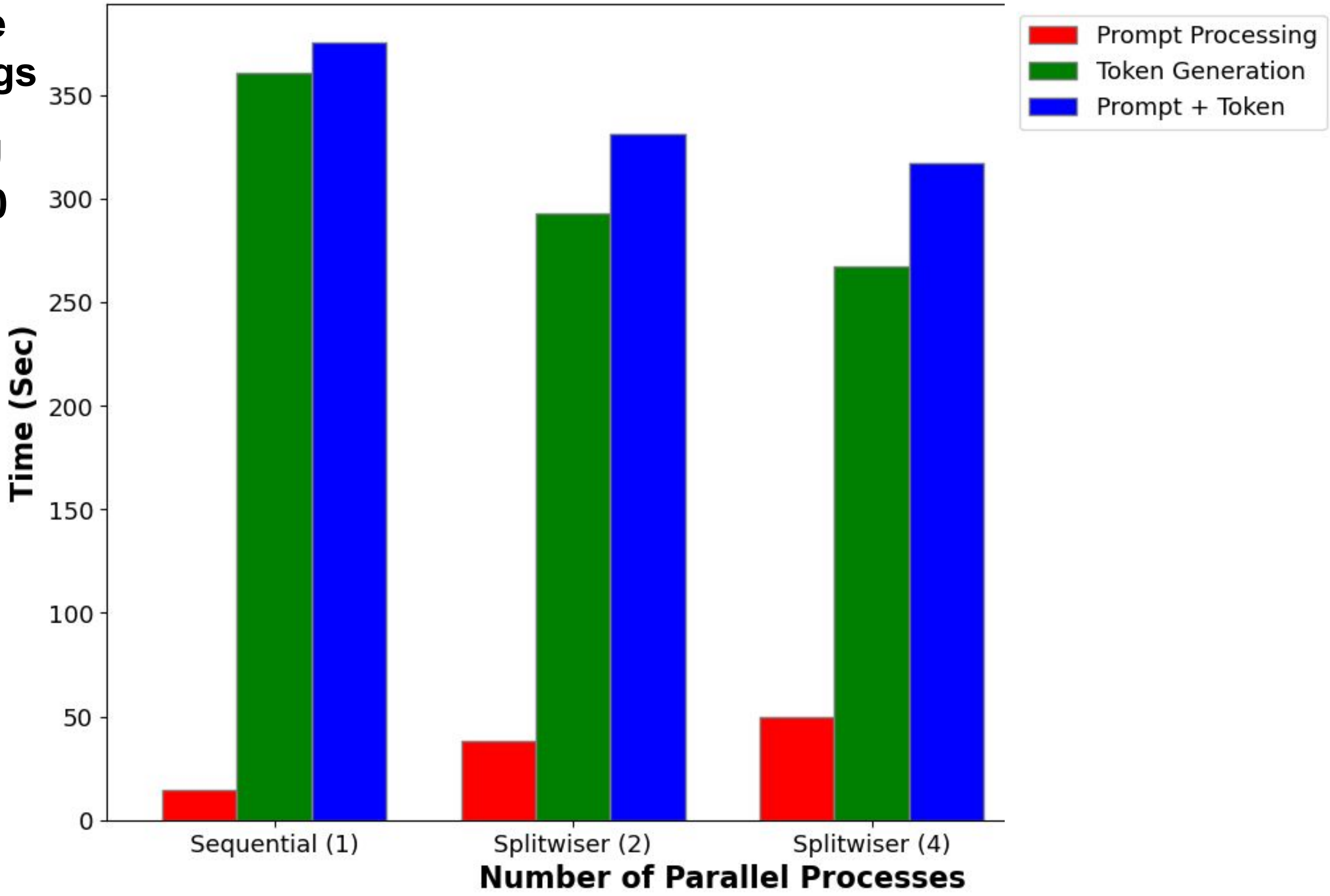
Time Savings

GPU A100



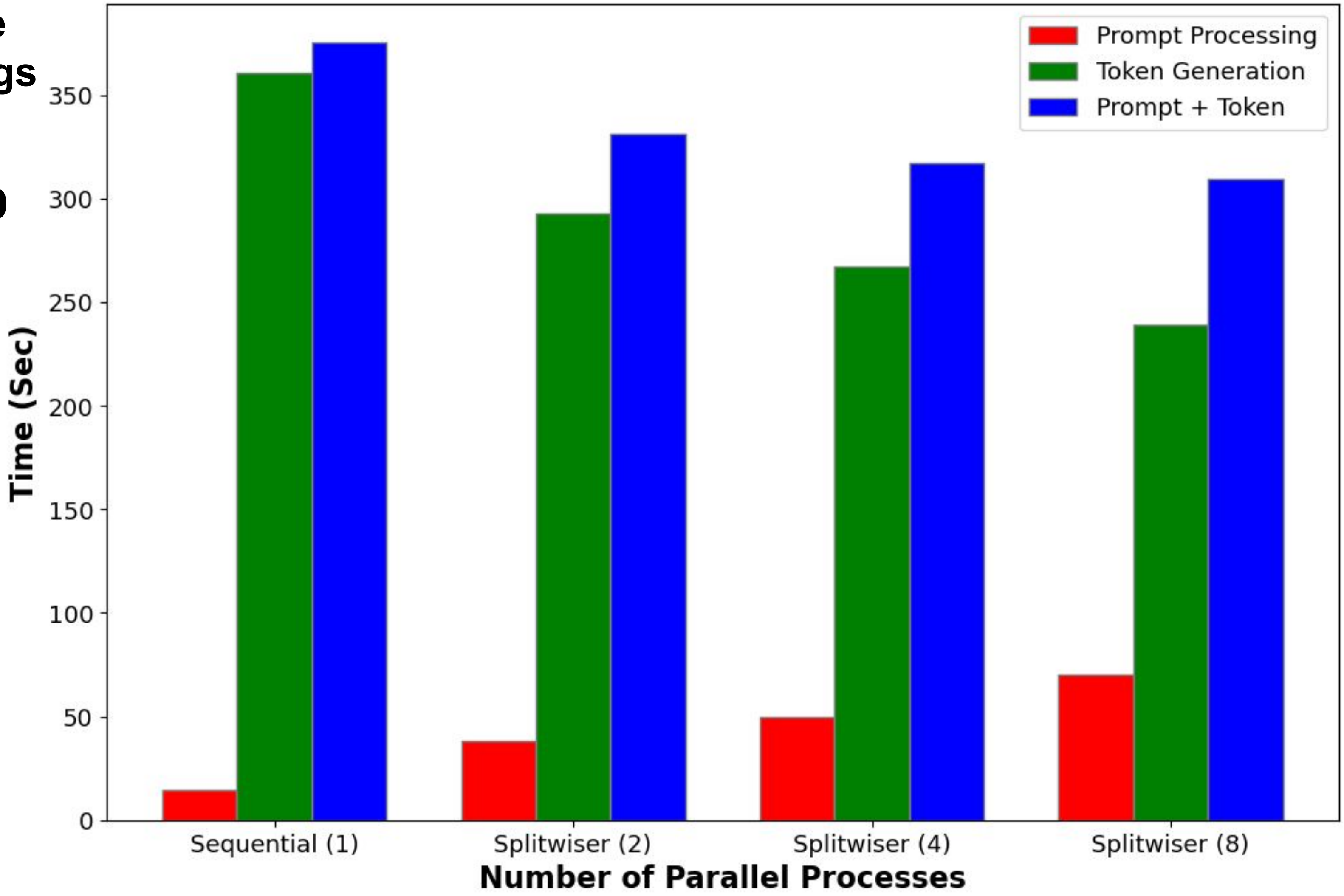
Time Savings

GPU A100



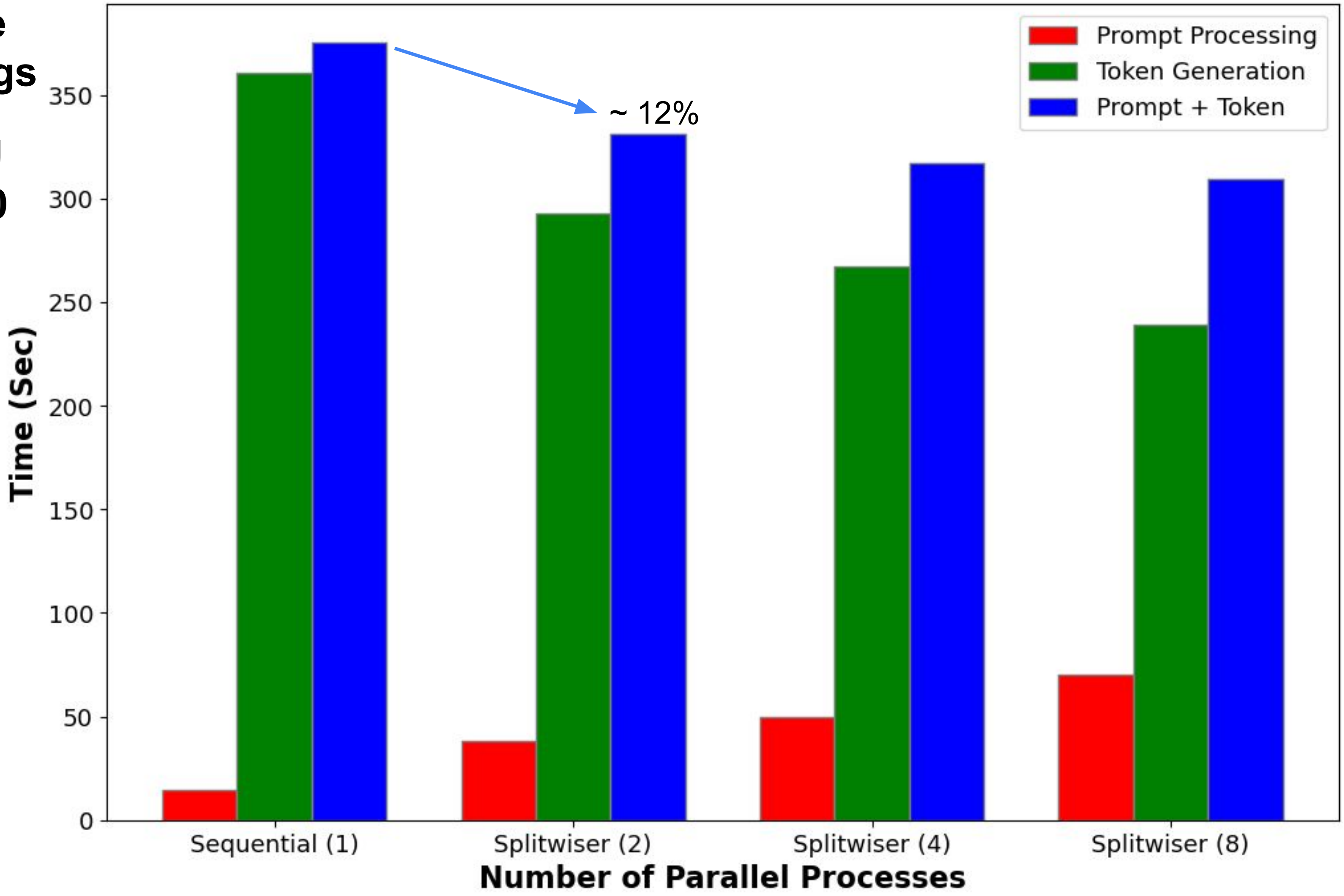
Time Savings

GPU A100



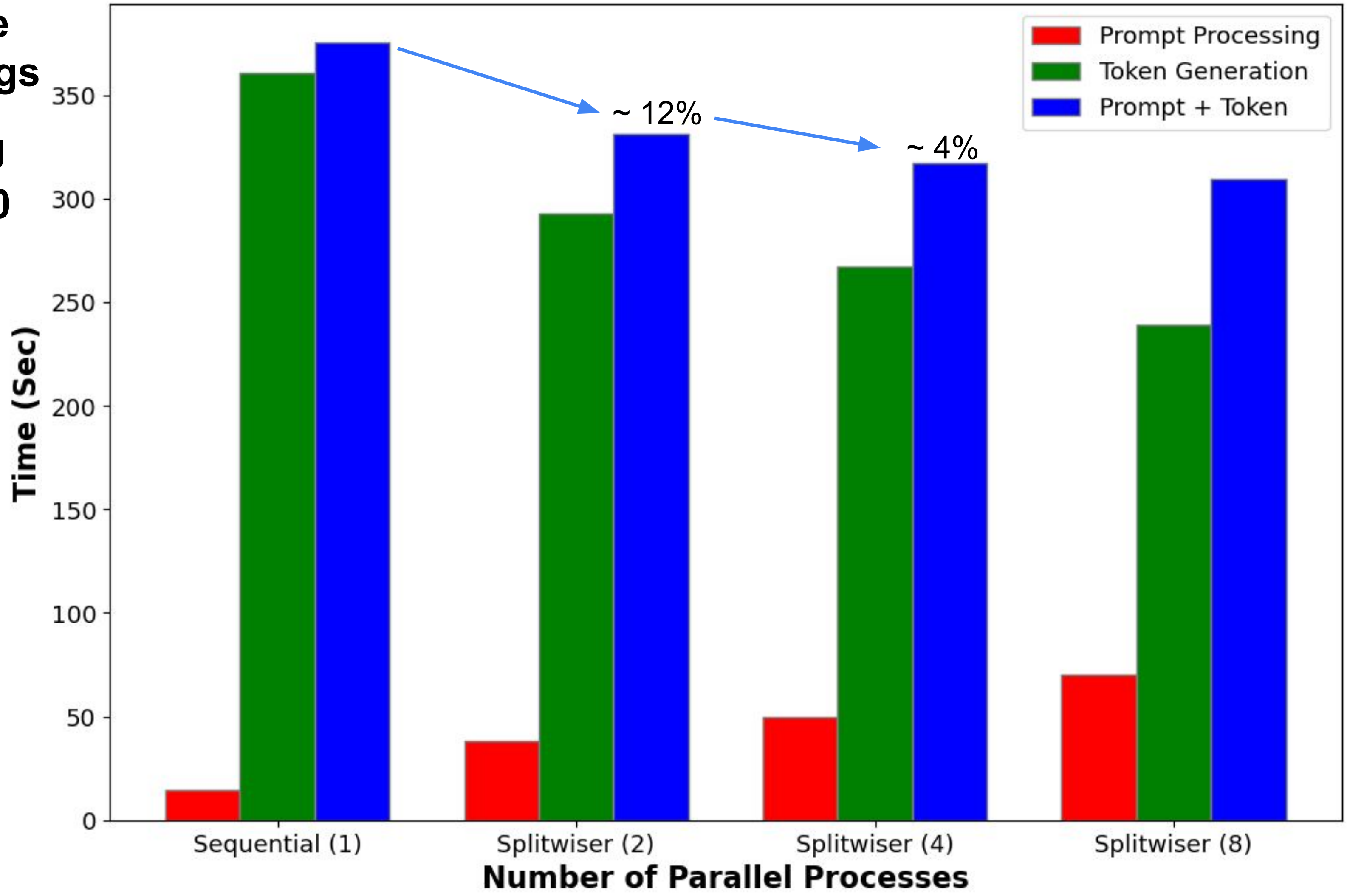
Time Savings

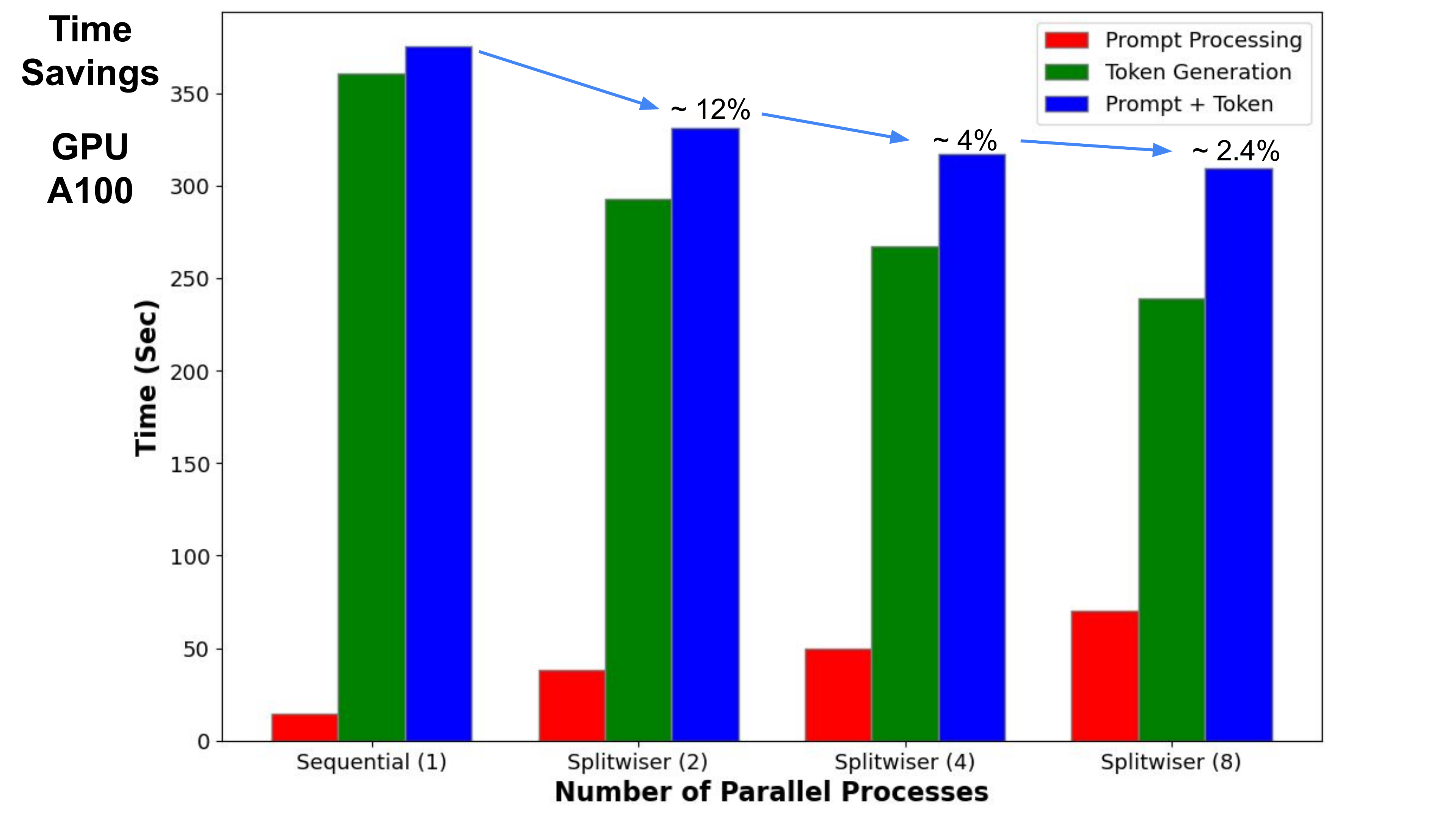
GPU A100

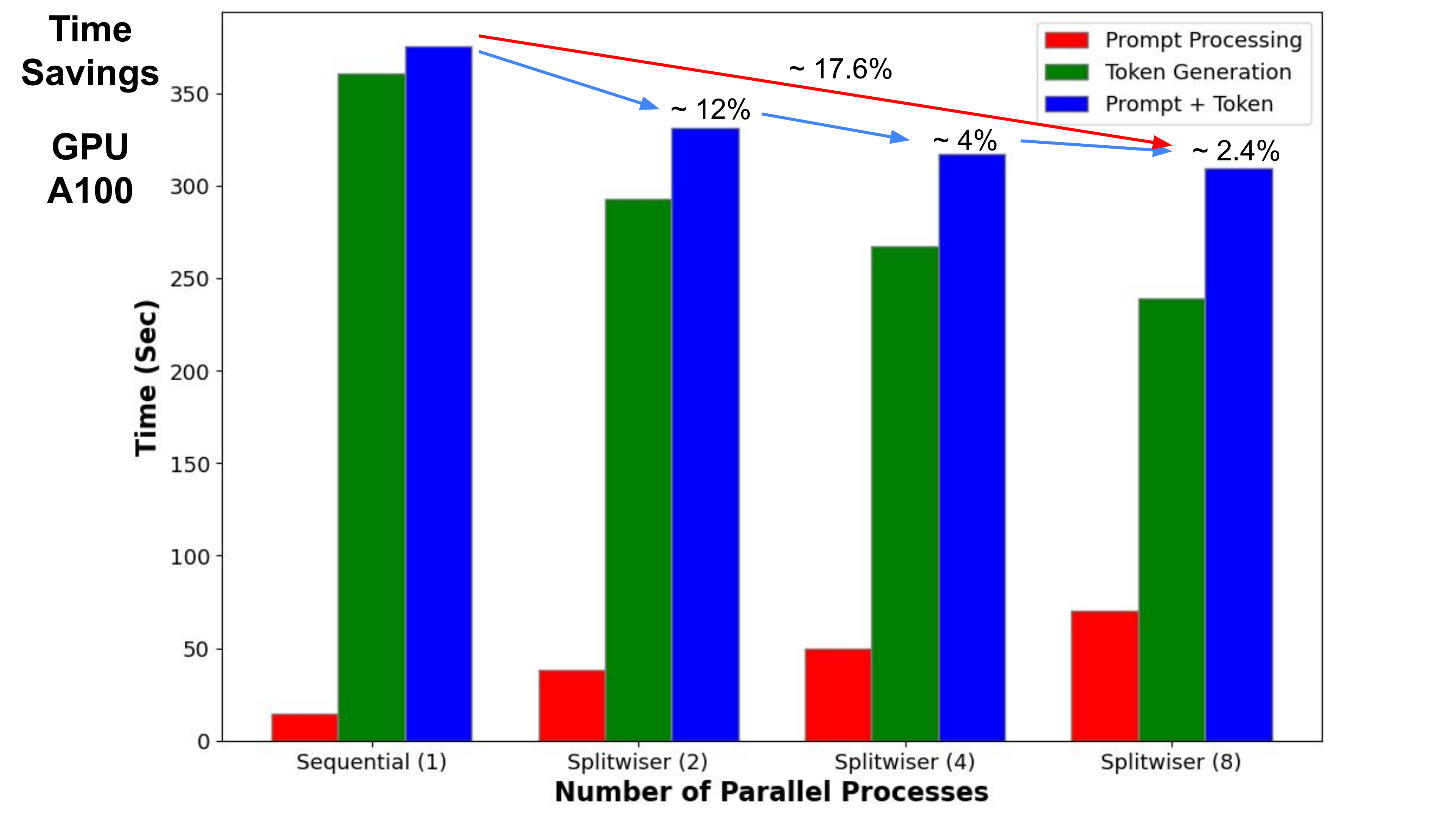


Time Savings

GPU A100



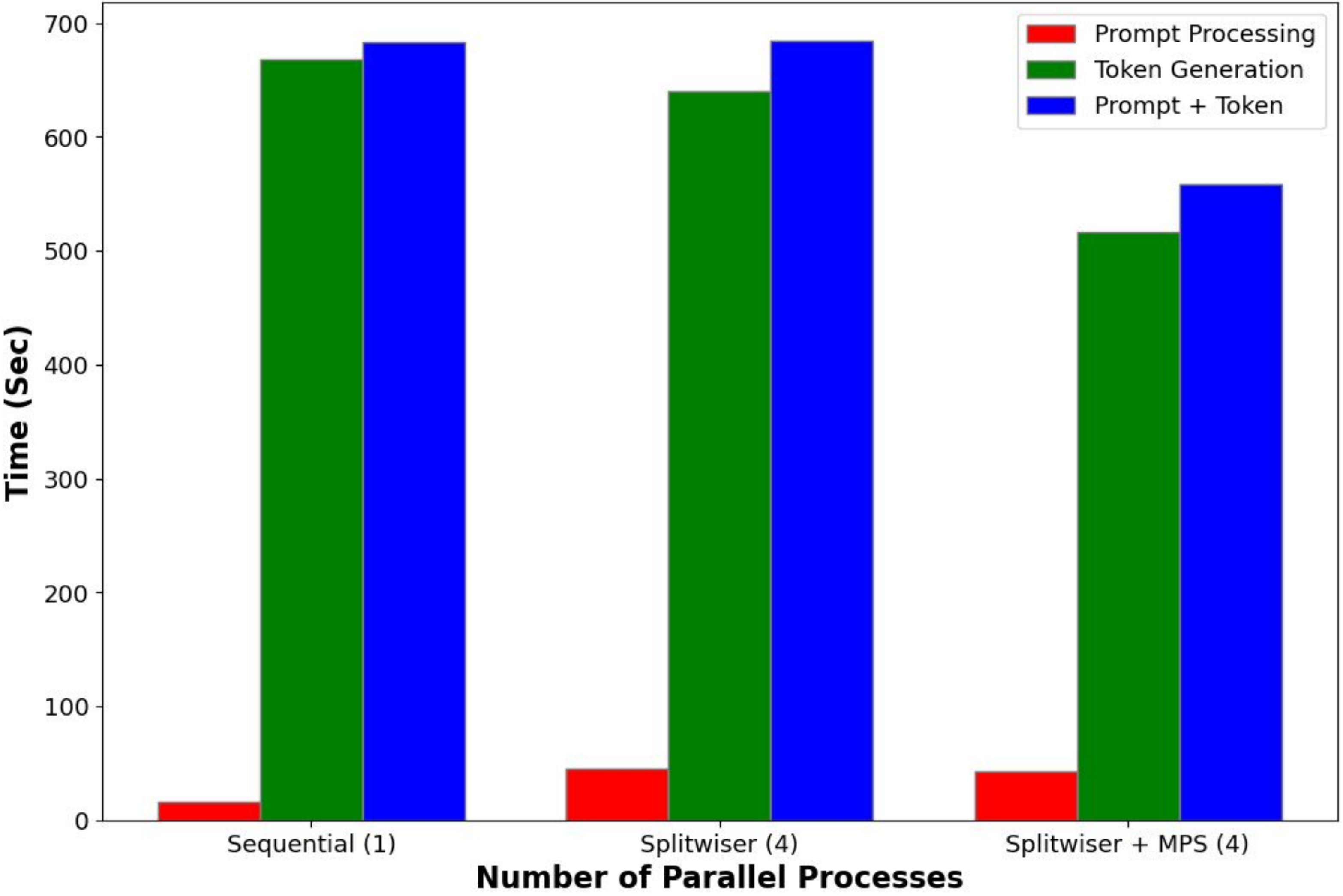

Time Savings
GPU A100
Prompt Processing
Token Generation
Prompt + Token
~ 12%
~ 4%
~ 2.4%
Time (Sec)
0
50
100
150
200
250
300
350
Sequential (1)
Splitwiser (2)
Splitwiser (4)
Splitwiser (8)
Number of Parallel Processes



Time Savings
GPU A100
Prompt Processing
Token Generation
Prompt + Token
~ 17.6%
~ 12%
~ 4%
~ 2.4%
Time (Sec)
0
50
100
150
200
250
300
350
Sequential (1)
Splitwiser (2)
Splitwiser (4)
Splitwiser (8)
Number of Parallel Processes

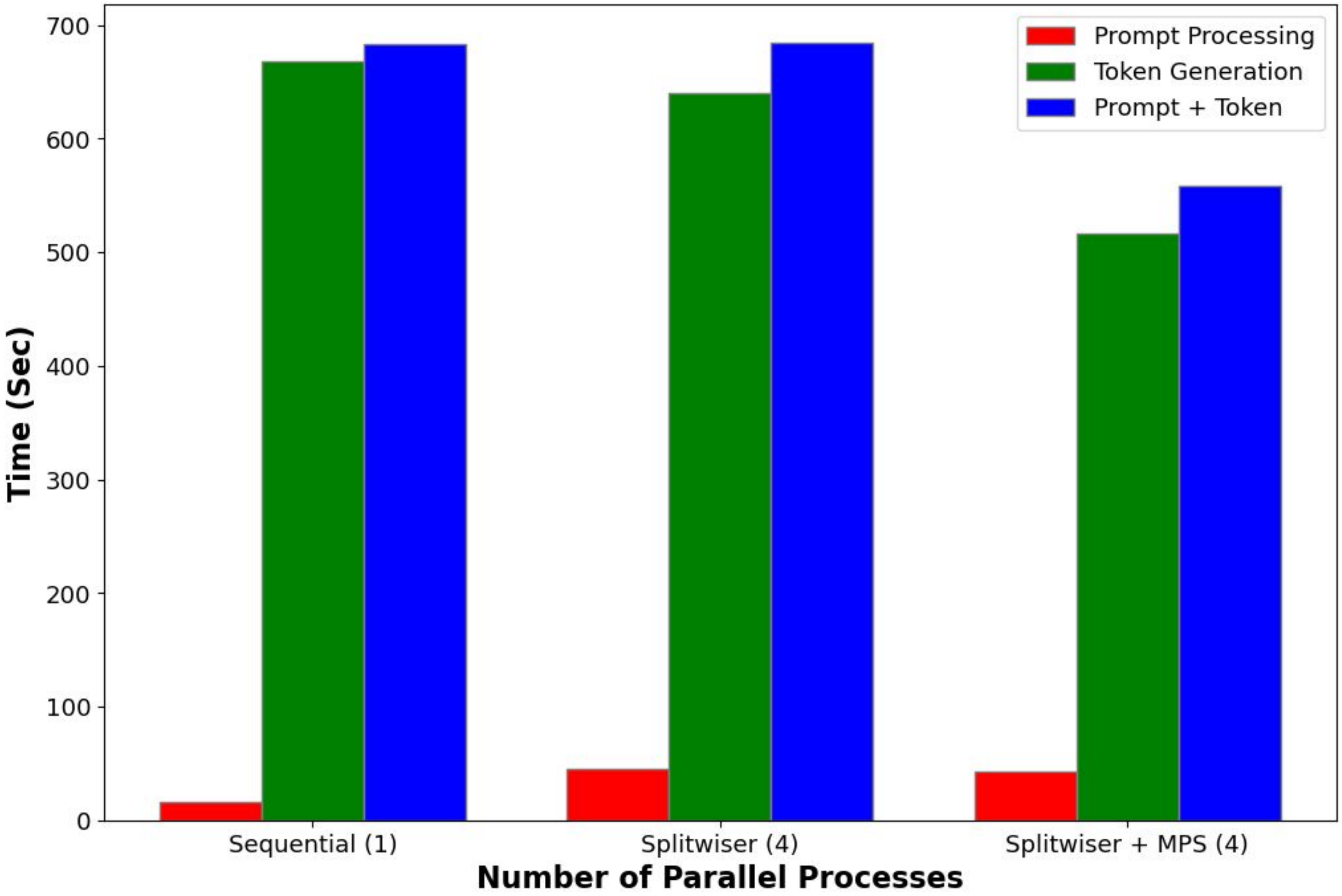
Time Savings

GPU A10



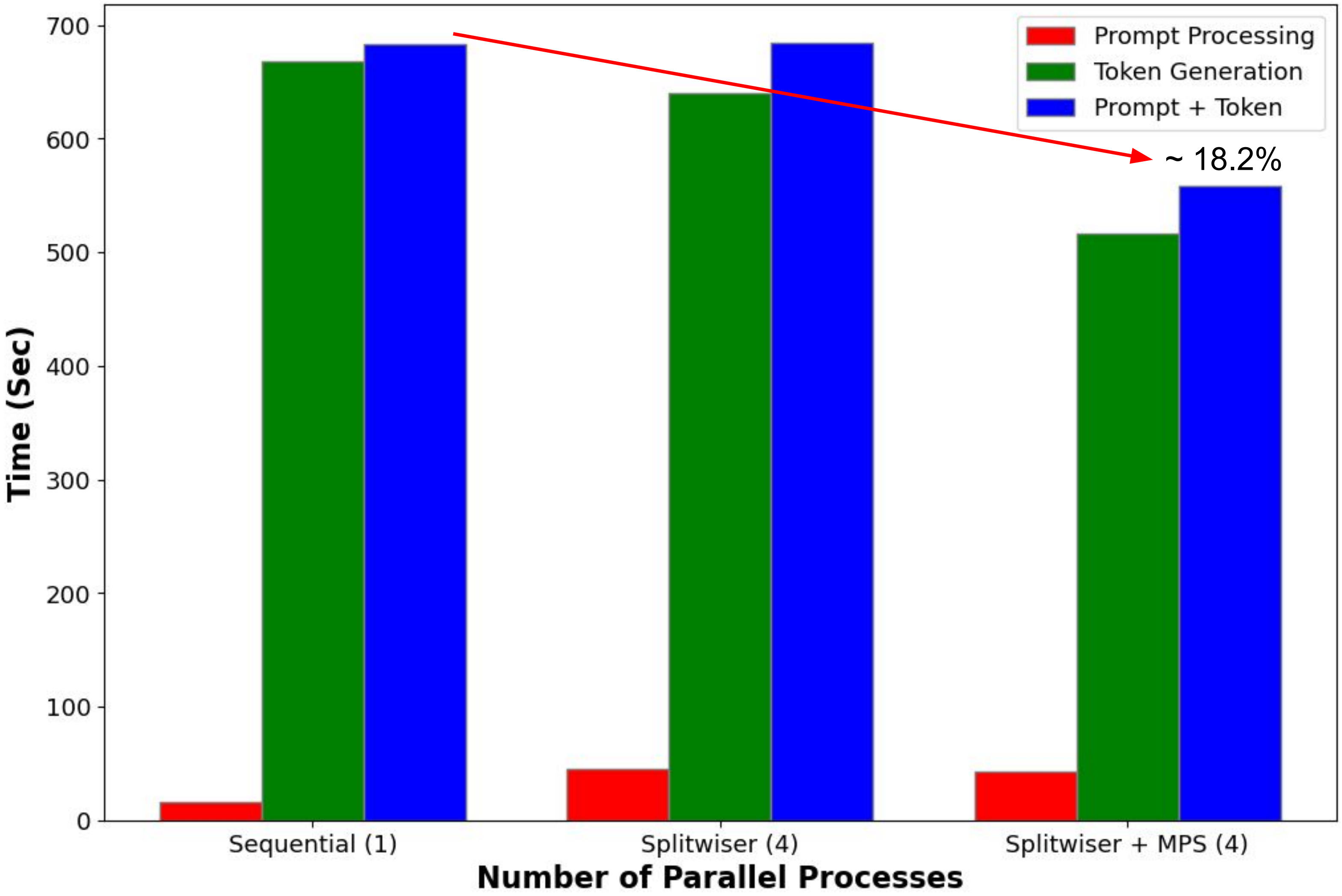
Time Savings

GPU A10



Time Savings

GPU A10

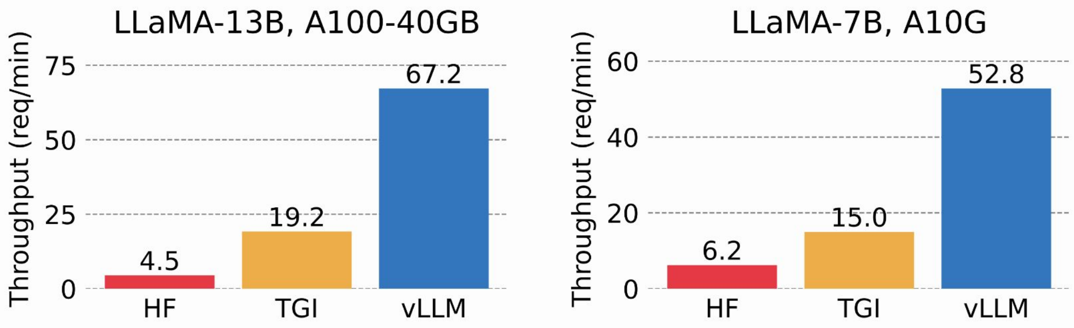


Solution Attempt 2B - vLLM + MP

Experiments and Results

vLLM Overview

- **What**: vLLM is the **Python library for LLM inference servicing** which the original Splitwise paper extends from
- **Why**: vLLM **greatly speeds up** multiple inference request servicing with techniques such as **PagedAttention** and **Continuous Batching**



Serving throughput when each request asks for *three parallel output completions*. vLLM achieves 8.5x - 15x higher throughput than HF and 3.3x - 3.5x higher throughput than TGI.

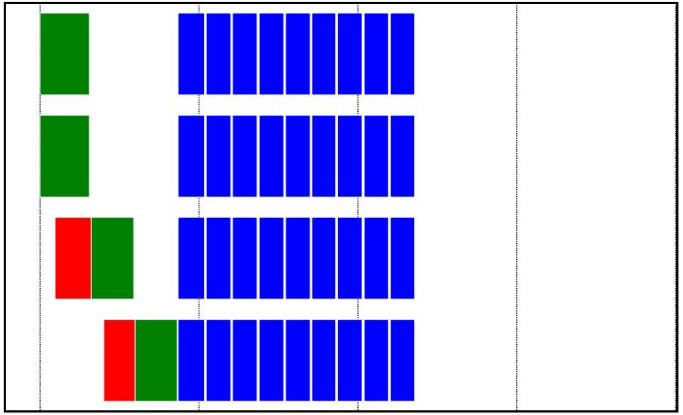
vLLM Scheduler

Each Step:

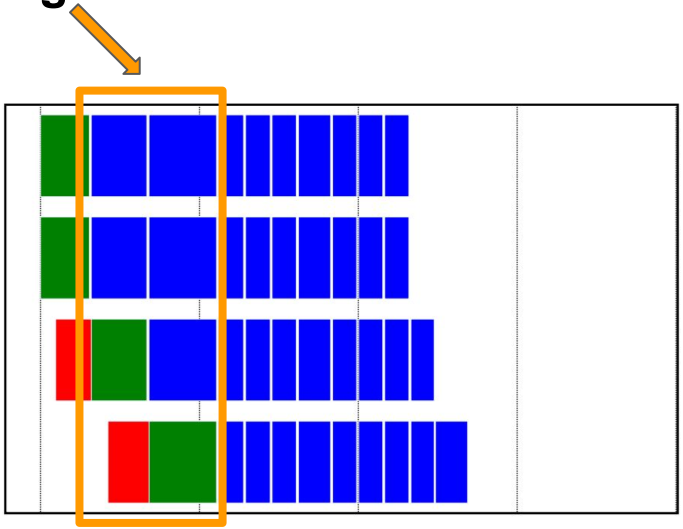
	Run Prompt Phase	OR	Run Token Phase
Schedule	Schedule batch of requests in **prompt phase (1 per req.)**		Schedule batch of requests in **token phase (N per req.)**
Pre-process	Per request: Fetch/process input tokens		Per request: Fetch/process KV cache
	Merge requests’ inputs into single set of input tensors		Merge requests’ inputs into single set of input tensors
Compute	Process merged tensors through LLM Sampler		Process merged tensors through LLM Sampler
Post-process	Separate Sampler output per request		Separate Sampler output per request

vLLM Potential Improvement

- vLLM currently implements **Continuous Batching**, but throughput could be further maximized with **Mixed Batching**



Continuous



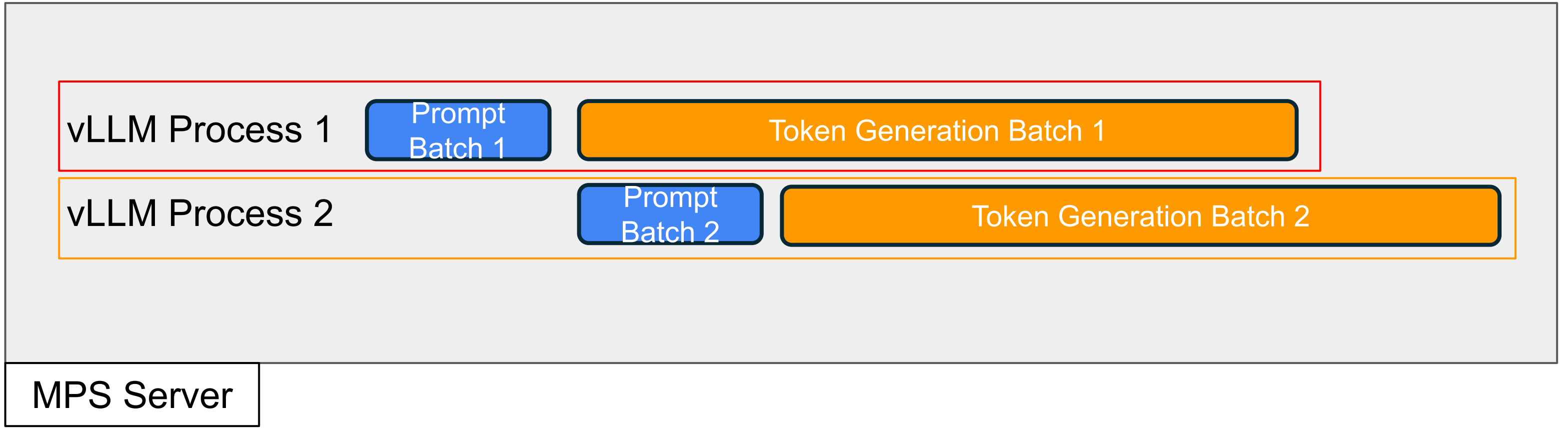
Mixed

vLLM Experiments Setup

- Model: opt-125m
- Input token size: 1024
- Output token size: 1024
- GPU: NVIDIA A10
- Batch Size: [10, 20, 40, 80, 160]

Attempt #1: vLLM + Multiprocessing

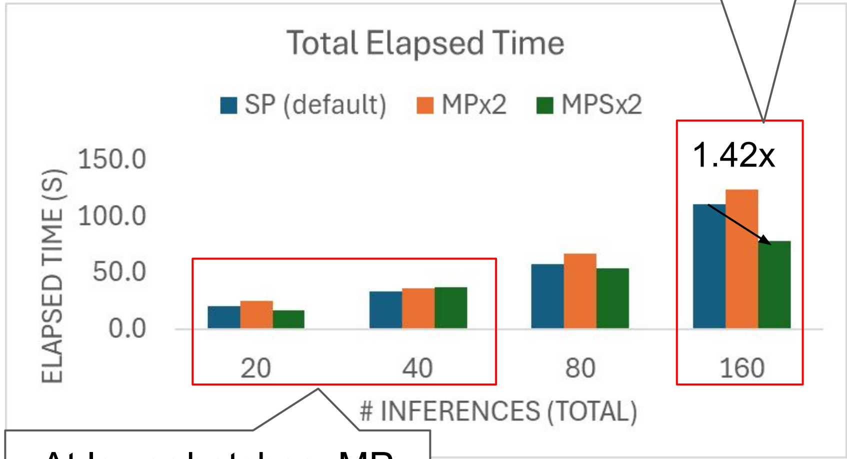
- Idea: Similar to previous Hugging Face + MP approach, run separate inference batches on separate processes to obtain parallelism
- Implementation:
  - MPx2: Instantiate a shared model, spawn 2 processes each running vLLM using the shared model
  - MPSx2: Same as above, but each process is a MPS client



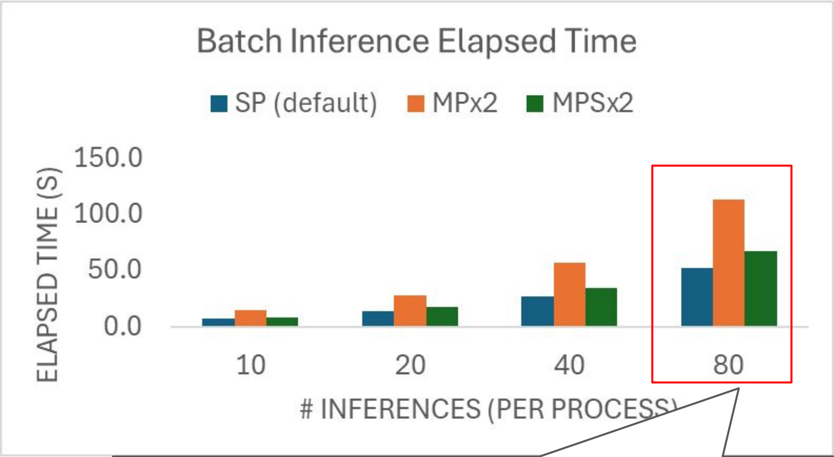
Attempt #1: vLLM + Multiprocessing

- Results:

Just MP is slower, but enabling MPS we get 1.42x speedup



At lower batches, MP hit w/ initialization overhead



MP(S) process will take longer to process its batch than SP w/ same batch size

Attempt #1: vLLM + Multiprocessing

- MP vs MPS: Possibly the benefit of MP throughput is lost from GPU context switching overhead, thus w/ MPS the latency is reduced
- Minimal src code modifications (shared model)
- Not scalable: # processes and max batch size will be limited by GPU memory (more obvious using larger model like llama2-7b)
  - vLLM running on single process has whole context of GPU device usage and scheduler designed to maximize accordingly
  - Can explore MP at lower-level, the vLLM scheduler…

Attempt #2: vLLM Scheduler + Multiprocessing

	Run Prompt Phase	**AND** Run Token Phase
Schedule	Schedule batch of requests in **prompt phase (1 per req.)**	Schedule batch of requests in **token phase (N per req.)**
Pre-process	Per request: Fetch/process input tokens Merge requests' inputs into single set of input tensors	Per request: Fetch/process KV cache Merge requests' inputs into single set of input tensors
Compute	Process merged tensors through LLM Sampler	Process merged tensors through LLM Sampler
Post-process	Separate Sampler output per request	Separate Sampler output per request

Attempt #2: vLLM Scheduler + Multiprocessing

- Implementation:
  - Modify scheduler to schedule both prompt and token phase batches
  - Spawn 2nd process to process prompt phase when both phases scheduled
  - Remove swapping (due to process spawning issues)
  - Remove CUDA graphs usage (due to process spawning issues)
- Result: Process spawning results in significant overhead/complications from replicating parent process objects. Not reasonable to create process on-demand.

Future Attempts: vLLM Scheduler (+MP?)

1. Instantiate the on-demand prompt process only once and use queues to pass inputs/outputs/required updates (block tables to locate data in memory)
   a. This second process will look like a slimmed down client vLLM that is only interacting with the queues to main vLLM process for processing jobs
   b. However, must be careful of communication/synchronization overhead: main process should continue working asynchronously if communication pending
2. Extending attempt #1: Write a scheduler that manages load between multiple vLLM processes
3. The only phase-specific processing step is **pre-processing**. Investigate if there's an efficient solution to merge the pre-processing of both the token and prompt phases.

Proposal #2: vLLM Mixed Scheduler (no MP)

	Run Prompt Phase	**AND**	Run Token Phase
Schedule	Schedule batch of requests in **prompt phase (1 per req.)**		Schedule batch of requests in **token phase (N per req.)**
Pre-process	Per request: Fetch/process input tokens/KV cache		
	Merge requests' inputs into single set of input tensors		
Compute	Process merged tensors through LLM Sampler		
Post-process	Separate Sampler output per request.		

Lessons Learned

- The recommended way to improve efficiency is to first maximize GPU kernels directly (better kernels, batch input for compute).
- Then, MP(S) can be explored at the next level of granularity such that you will likely have multiple kernels executing simultaneously
  - No gain if CPU doesn’t launch next kernel fast enough
  - If GPU utilization already high, CUDA schedules kernels sequentially
  - You only see the benefit of MP once you run it for a large number of inference steps
    - The benefit of improvement in steady-state throughput outweighs the cost of initial MP overhead over large number of iterations
- Applying MP to integrate with a developed scheduler/resource manager like vLLM is non-trivial

Thank You!