

zoo-bench performance report

raw data: results/zoo-framework-0.11.0/20261011T093730Z.json

Run environment

item	value
CPU model	AMD EPYC 7763 64-Core Processor
logical cores	4
platform	Linux-x86_64
operating system	Linux 6.17.0-1022-azure (#22-Ubuntu SMP Mon Jul 27 17:24:03 UTC 2026)
Python	CPython 3.13.16
interpreter	/opt/hostedtoolcache/Python/3.13.16/x64/bin/python
GIL	default
framework under test (distribution metadata)	0.11.0
framework under test (module __version__)	0.11.0 (footnote only, not the version criterion)
drive-surface generation of the framework under test	current-generation drive surface (scheduling core + scheduling model; the completion signal comes from the result reactor)
harness	zoo-bench 0.0.1 @ 9a42be24ac43f75e2e5d7b16434f91ba3d0717b6
command to reproduce	zoo-bench run --framework zoo-framework==0.11.0

dist_version (the distribution metadata) is the source of truth; module_version is only a footnote

Workload

- composition: one unit of work = JSON serialize + deserialize + string concatenation, repeated until the target duration is reached; the tiers are declared by body_tiers_us in matrix.yaml
- **the workload is a stand-in, not a real trace**: its shape resembles business actions such as 'take a frame of data -> serialize -> parse -> transform strings', but this repository holds no real trace, so every conclusion is conditional on that

Measurement methodology

- **the body duration is instrumented within the same run**: the body measures itself internally and reports back; framework overhead = end-to-end - it. **No cross-run subtraction** - two runs differ in state (caches, frequencies, scheduling noise), so subtracting introduces systematic bias.
- **warmups are excluded from statistics**: each unit first runs 3 warmup rounds that are discarded, then only the following 30 measured rounds are counted; the sample count equals 'measured rounds x concurrency'.
- **absolute durations are not comparable across runs**: they only show the order of magnitude; across versions/machines read the **relative quantities within the same run** (overhead share, multiples against the baselines), which are insensitive to overall speed.

Conclusion summary

- Once the framework under test zoo runs a body of about 2700 microseconds or more, its framework overhead drops below 15% of end-to-end; below that tier, the main price of choosing it is the framework overhead itself.

- At concurrency 1 and a 40 microsecond body, the framework under test is 2.14x faster than process_pool (the fastest measured tier at that concurrency).
- At concurrency 1, once the body exceeds about 40 microseconds, apscheduler is no longer faster than the framework under test.
- At concurrency 4, once the body exceeds about 40 microseconds, apscheduler is no longer faster than the framework under test.
- At concurrency 16, once the body exceeds about 40 microseconds, apscheduler is no longer faster than the framework under test.
- At concurrency 64, once the body exceeds about 40 microseconds, apscheduler is no longer faster than the framework under test.
- At concurrency 1, within the measured tiers asyncio_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 4, once the body exceeds about 10000 microseconds, asyncio_pool is no longer faster than the framework under test.
- At concurrency 16, within the measured tiers asyncio_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 64, within the measured tiers asyncio_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 1, within the measured tiers bare_thread is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 4, within the measured tiers bare_thread is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 16, once the body exceeds about 40 microseconds, bare_thread is no longer faster than the framework under test.
- At concurrency 64, within the measured tiers bare_thread is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 1, once the body exceeds about 40 microseconds, process_pool is no longer faster than the framework under test.
- At concurrency 4, within the measured tiers process_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 16, within the measured tiers process_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 64, once the body exceeds about 40 microseconds, process_pool is no longer faster than the framework under test.
- At concurrency 1, within the measured tiers thread_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 4, within the measured tiers thread_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 16, within the measured tiers thread_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 64, within the measured tiers thread_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.

a single-point speedup has no selection meaning; the conclusion is phrased as 'which scheme to choose at what body duration' (the same framework overhead is 70% of a 40 microsecond task and a few percent of a 10

millisecond one)

Overhead threshold crossings

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
apscheduler	1	15.00%	2700	40, 300, 2700, 10000	this is the smallest measured tier that satisfies the condition; the true crossing point lies between it and the previous tier
apscheduler	4	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
apscheduler	16	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.
apscheduler	64	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
asyncio_pool	1	15.00%	2700	40, 300, 2700, 10000	this is the smallest measured tier that satisfies the condition; the true crossing point lies between it and the previous tier
asyncio_pool	4	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
asyncio_pool	16	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.
asyncio_pool	64	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
bare_thread	1	15.00%	2700	40, 300, 2700, 10000	this is the smallest measured tier that satisfies the condition; the true crossing point lies between it and the previous tier
bare_thread	4	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
bare_thread	16	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.
bare_thread	64	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
process_pool	1	15.00%	2700	40, 300, 2700, 10000	this is the smallest measured tier that satisfies the condition; the true crossing point lies between it and the previous tier
process_pool	4	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
process_pool	16	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.
process_pool	64	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
thread_pool	1	15.00%	300	40, 300, 2700, 10000	this is the smallest measured tier that satisfies the condition; the true crossing point lies between it and the previous tier
thread_pool	4	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
thread_pool	16	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.
thread_pool	64	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
zoo	1	15.00%	2700	40, 300, 2700, 10000	this is the smallest measured tier that satisfies the condition; the true crossing point lies between it and the previous tier
zoo	4	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
zoo	16	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.
zoo	64	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

Relative turning points

baseline	concurrency	first tier where it is no longer faster (microseconds)	note
apscheduler	1	40	this is the smallest measured tier that satisfies the condition
apscheduler	4	40	this is the smallest measured tier that satisfies the condition
apscheduler	16	40	this is the smallest measured tier that satisfies the condition
apscheduler	64	40	this is the smallest measured tier that satisfies the condition
asyncio_pool	1	-	not observed: within the measured tiers this baseline is always faster than the framework under test
asyncio_pool	4	10000	this is the smallest measured tier that satisfies the condition
asyncio_pool	16	-	not observed: within the measured tiers this baseline is always faster than the framework under test
asyncio_pool	64	-	not observed: within the measured tiers this baseline is always faster than the framework under test
bare_thread	1	-	not observed: within the measured tiers this baseline is always faster than the framework under test
bare_thread	4	-	not observed: within the measured tiers this baseline is always faster than the framework under test
bare_thread	16	40	this is the smallest measured tier that satisfies the condition
bare_thread	64	-	not observed: within the measured tiers this baseline is always faster than the framework under test
process_pool	1	40	this is the smallest measured tier that satisfies the condition
process_pool	4	-	not observed: within the measured tiers this baseline is always faster than the framework under test

baseline	concurrency	first tier where it is no longer faster (microseconds)	note
process_pool	16	-	not observed: within the measured tiers this baseline is always faster than the framework under test
process_pool	64	40	this is the smallest measured tier that satisfies the condition
thread_pool	1	-	not observed: within the measured tiers this baseline is always faster than the framework under test
thread_pool	4	-	not observed: within the measured tiers this baseline is always faster than the framework under test
thread_pool	16	-	not observed: within the measured tiers this baseline is always faster than the framework under test
thread_pool	64	-	not observed: within the measured tiers this baseline is always faster than the framework under test

Charts

[missing figure: figures/en-overhead_ratio.svg]

Body tiers at concurrency 4 only; ratios hold within a single run. The overhead numbers of concurrency 16, 64 were withheld (beyond this machine's parallelism; see the caliber section) and are not charted

[missing figure: figures/en-throughput.svg]

Body tier 40 microseconds only; throughput holds within a single run

[missing figure: figures/en-relative_multiple.svg]

One panel per measured concurrency (4 in total); multiples hold within a single run

Dimension: Latency percentiles and jitter

end-to-end / concurrency; percentiles and relative spread are given in the same table - a metric that only gives the median is not acceptable

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
1	40	157.03 microseconds	308.03 microseconds	144.14 microseconds	141.61 microseconds	336.29 microseconds	79.31 microseconds
1	300	412.58 microseconds	562.94 microseconds	389.15 microseconds	393.51 microseconds	591.15 microseconds	325.37 microseconds
1	2700	2.778 milliseconds	2.948 milliseconds	2.663 milliseconds	2.725 milliseconds	3.013 milliseconds	2.623 milliseconds
1	10000	9.929 milliseconds	10.112 milliseconds	9.399 milliseconds	9.630 milliseconds	10.125 milliseconds	9.369 milliseconds

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
4	40	177.06 microseconds	257.17 microseconds	156.44 microseconds	162.35 microseconds	176.27 microseconds	94.18 microseconds
4	300	442.41 microseconds	546.38 microseconds	414.86 microseconds	411.77 microseconds	331.48 microseconds	348.68 microseconds
4	2700	2.878 milliseconds	2.989 milliseconds	2.729 milliseconds	2.648 milliseconds	1.400 milliseconds	2.702 milliseconds
4	10000	9.947 milliseconds	10.095 milliseconds	9.955 milliseconds	9.618 milliseconds	4.617 milliseconds	9.839 milliseconds
16	40	158.98 microseconds	230.76 microseconds	136.04 microseconds	168.76 microseconds	158.72 microseconds	90.57 microseconds
16	300	440.91 microseconds	512.03 microseconds	395.94 microseconds	427.79 microseconds	276.84 microseconds	355.80 microseconds
16	2700	2.885 milliseconds	2.952 milliseconds	2.725 milliseconds	2.700 milliseconds	1.370 milliseconds	2.742 milliseconds
16	10000	9.988 milliseconds	10.097 milliseconds	9.980 milliseconds	9.717 milliseconds	4.698 milliseconds	9.867 milliseconds
64	40	194.80 microseconds	241.67 microseconds	124.21 microseconds	171.85 microseconds	202.33 microseconds	90.47 microseconds
64	300	452.02 microseconds	510.06 microseconds	378.92 microseconds	427.59 microseconds	296.22 microseconds	341.71 microseconds
64	2700	2.928 milliseconds	2.995 milliseconds	2.779 milliseconds	2.705 milliseconds	1.357 milliseconds	2.705 milliseconds
64	10000	10.517 milliseconds	10.128 milliseconds	10.259 milliseconds	9.718 milliseconds	4.709 milliseconds	10.415 milliseconds

per-unit values (percentiles, spread, raw per-round samples) are in 'Appendix: all values' at the end.

Dimension: Framework overhead share

overhead = end-to-end / concurrency - the body duration measured within the same run (no cross-run subtraction). **cells with -**: that concurrency exceeds this machine's parallel capacity; the body's self-reported duration contains scheduling waits from oversubscription and is not comparable with the row's end-to-end per task, so no overhead figure is given (end-to-end and throughput live in their own dimensions)

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
1	40	68.65%	81.31%	67.95%	63.22%	83.79%	39.28%
1	300	27.69%	46.45%	26.86%	24.74%	47.86%	11.59%
1	2700	5.00%	9.43%	4.87%	4.28%	11.72%	1.96%
1	10000	1.74%	4.37%	1.50%	1.20%	3.66%	0.60%
4	40	72.38%	77.32%	66.81%	68.24%	46.48%	48.94%
4	300	31.63%	42.90%	25.61%	29.06%	-	15.26%
4	2700	5.40%	10.29%	5.57%	4.38%	-	4.06%
4	10000	-	-	-	-	-	-
16	40	-	-	-	-	-	-
16	300	-	-	-	-	-	-
16	2700	-	-	-	-	-	-
16	10000	-	-	-	-	-	-

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
64	40	-	-	-	-	-	-
64	300	-	-	-	-	-	-
64	2700	-	-	-	-	-	-
64	10000	-	-	-	-	-	-

per-unit values (percentiles, spread, raw per-round samples) are in 'Appendix: all values' at the end.

Dimension: Throughput and concurrency scaling

throughput = concurrency / median end-to-end; whether it keeps rising with concurrency is the scaling behaviour

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
1	40	6368.3	3246.5	6937.9	7061.4	2973.7	12609.0
1	300	2423.8	1776.4	2569.7	2541.2	1691.6	3073.4
1	2700	359.9	339.2	375.5	367.0	331.9	381.2
1	10000	100.7	98.9	106.4	103.8	98.8	106.7
4	40	22591.2	15554.0	25569.1	24638.4	22692.6	42474.1
4	300	9041.4	7321.0	9641.7	9714.1	12067.1	11472.0
4	2700	1389.7	1338.2	1465.6	1510.8	2856.9	1480.5
4	10000	402.1	396.3	401.8	415.9	866.4	406.6
16	40	100640.5	69336.7	117614.4	94809.1	100804.1	176656.6
16	300	36288.7	31248.4	40410.1	37401.2	57794.7	44969.3
16	2700	5545.8	5419.6	5872.3	5926.5	11681.1	5835.0
16	10000	1601.9	1584.6	1603.2	1646.6	3405.5	1621.6
64	40	328542.1	264822.7	515270.2	372428.2	316315.6	707442.1
64	300	141586.6	125476.2	168900.1	149676.6	216054.3	187293.3
64	2700	21859.0	21372.5	23028.2	23658.9	47176.6	23661.8
64	10000	6085.3	6319.2	6238.4	6585.6	13590.5	6144.8

per-unit values (percentiles, spread, raw per-round samples) are in 'Appendix: all values' at the end.

Dimension: Overhead attribution

scope: splits the per-task end-to-end into submit side / handoff / body / return and, at the same tier and concurrency, **compares each segment against every baseline**, answering 'which segment the excess falls on'. It is **diagnostic, not the main evidence**: it samples a small set of tiers, and does not compare total overhead against the baselines. **A negative handoff is not an error**: it means the submit call had not returned yet while the body was already running - schemes that count thread creation as submit side behave this way (measured: bare_thread handoff -343 microseconds). **The return segment includes waiting for the completion signal** (the last stretch before draining returns); it shares the main measurement's scope

status: ok

this dimension is a **diagnosis**, not the main evidence: the sampled tiers and concurrencies are a fixed small batch, not the whole matrix. The concurrency ceiling is the machine's parallel capacity - beyond it, what is measured is contention;

sampled tiers (microseconds): 40, 10000; concurrency: [1, 4]; this machine's parallel capacity: 4

scheme	concurrency	tier (microseconds)	submit side	handoff	body	return	status
apscheduler	1	40	105.70 microseconds	89.95 microseconds	51.93 microseconds	39.97 microseconds	ok
apscheduler	4	40	88.92 microseconds	519.14 microseconds	58.30 microseconds	235.85 microseconds	ok
apscheduler	1	10000	190.14 microseconds	131.76 microseconds	9.755 milliseconds	61.09 microseconds	ok
apscheduler	4	10000	123.24 microseconds	10.454 milliseconds	14.656 milliseconds	9.685 milliseconds	ok
asyncio_pool	1	40	17.51 microseconds	26.17 microseconds	47.18 microseconds	57.79 microseconds	ok
asyncio_pool	4	40	19.75 microseconds	261.66 microseconds	53.31 microseconds	295.26 microseconds	ok
asyncio_pool	1	10000	25.81 microseconds	39.08 microseconds	9.477 milliseconds	95.70 microseconds	ok
asyncio_pool	4	10000	23.73 microseconds	12.437 milliseconds	14.392 milliseconds	9.922 milliseconds	ok
bare_thread	1	40	143.92 microseconds	-77.44 microseconds	52.54 microseconds	30.62 microseconds	ok
bare_thread	4	40	153.77 microseconds	-75.41 microseconds	51.61 microseconds	337.75 microseconds	ok
bare_thread	1	10000	5.201 milliseconds	-5092.36 microseconds	9.693 milliseconds	48.96 microseconds	ok
bare_thread	4	10000	5.227 milliseconds	-5084.12 microseconds	14.960 milliseconds	9.964 milliseconds	ok
process_pool	1	40	-	-	-	-	not_measurable: the body's start/end timestamps are unavailable - it did not run in this process (cross-process units cannot be split this way)

scheme	concurrency	tier (microse conds)	submit side	handoff	body	return	status
process_pool	4	40	-	-	-	-	not_measura ble: the body's start/end timestamps are unavailable - it did not run in this process (cros s-process units cannot be split this way)
process_pool	1	10000	-	-	-	-	not_measura ble: the body's start/end timestamps are unavailable - it did not run in this process (cros s-process units cannot be split this way)
process_pool	4	10000	-	-	-	-	not_measura ble: the body's start/end timestamps are unavailable - it did not run in this process (cros s-process units cannot be split this way)
thread_pool	1	40	8.44 microseconds	10.47 microseconds	44.63 microseconds	15.66 microseconds	ok
thread_pool	4	40	11.21 microseconds	178.51 microseconds	49.72 microseconds	151.75 microseconds	ok
thread_pool	1	10000	17.21 microseconds	20.60 microseconds	9.481 milliseconds	40.80 microseconds	ok
thread_pool	4	10000	13.85 microseconds	10.088 milliseconds	14.762 milliseconds	9.041 milliseconds	ok
zoo	1	40	64.50 microseconds	41.26 microseconds	56.29 microseconds	73.36 microseconds	ok
zoo	4	40	56.52 microseconds	303.27 microseconds	48.83 microseconds	249.76 microseconds	ok

scheme	concurrency	tier (microse conds)	submit side	handoff	body	return	status
zoo	1	10000	109.02 microseconds	38.07 microseconds	9.694 milliseconds	140.97 microseconds	ok
zoo	4	10000	86.25 microseconds	10.271 milliseconds	20.159 milliseconds	5.744 milliseconds	ok

where the excess over each baseline falls

baseline	concurrency	tier (microseconds)	excess per task	mostly on	gap in that segment
apscheduler	1	40	-57.52 microseconds	return	33.39 microseconds
asyncio_pool	1	40	87.51 microseconds	submit side	46.99 microseconds
bare_thread	1	40	85.04 microseconds	handoff	118.70 microseconds
thread_pool	1	40	156.15 microseconds	return	57.70 microseconds
apscheduler	4	40	-246.88 microseconds	return	13.91 microseconds
asyncio_pool	4	40	16.03 microseconds	handoff	41.61 microseconds
bare_thread	4	40	200.23 microseconds	handoff	378.67 microseconds
thread_pool	4	40	279.47 microseconds	handoff	124.76 microseconds
apscheduler	1	10000	-184.50 microseconds	return	79.88 microseconds
asyncio_pool	1	10000	324.83 microseconds	body	217.01 microseconds
bare_thread	1	10000	121.06 microseconds	handoff	5.130 milliseconds
thread_pool	1	10000	401.12 microseconds	body	212.71 microseconds
apscheduler	4	10000	1.529 milliseconds	body	5.503 milliseconds
asyncio_pool	4	10000	1.542 milliseconds	body	5.767 milliseconds
bare_thread	4	10000	8.639 milliseconds	handoff	15.355 milliseconds
thread_pool	4	10000	2.222 milliseconds	body	5.396 milliseconds

instrumentation seams used (temporarily wrapped on the harness side, no framework code changed):

BaseWaiter._dispatch_worker (handing a worker to the scheduling model);

WorkerDispatchCore.resolve_period / resolve_phase / resolve_run_timeout (per-round policy lookups);

BaseWaiter.execute_service (one scheduling round)

the cost of instrumentation itself (on the framework-under-test side, paired from two rounds in the same subprocess, so it can be discounted honestly): zoo 40 microseconds / concurrency 1: without instrumentation 230.24 microseconds vs with 236.18 microseconds, a difference of 5.94 microseconds; zoo 40 microseconds / concurrency 4: without instrumentation 660.66 microseconds vs with 661.52 microseconds, a difference of 0.85 microseconds; zoo 10000 microseconds / concurrency 1: without instrumentation 9.995 milliseconds vs with 9.969 milliseconds, a difference of 26.32 microseconds (faster with instrumentation: this round's machine

drift exceeded the instrumentation cost); zoo 10000 microseconds / concurrency 4: without instrumentation 41.278 milliseconds vs with 41.169 milliseconds, a difference of 109.45 microseconds (faster with instrumentation: this round's machine drift exceeded the instrumentation cost)

additivity of the sub-measurements: the sub-measurements are **independent measurements**: each one's duration contains its own preemption waits, so their sum can exceed the submit side they all live in (about 3% measured). Do not read them as an additive partition. measured overlap this round: zoo 40 microseconds / concurrency 4 overlaps 3.5%; zoo 10000 microseconds / concurrency 1 overlaps 18.9%; zoo 10000 microseconds / concurrency 4 overlaps 94.3%

what the submit side of the framework under test consists of

- concurrency 1, tier 40 microseconds: rest of the scheduling round (due checks, locks, list upkeep) 2.41 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 28.54 microseconds; per-round policy lookups (period / phase / timeout each read config) 23.88 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 9.79 microseconds
- concurrency 4, tier 40 microseconds: rest of the scheduling round (due checks, locks, list upkeep) 0.00 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 23.63 microseconds; per-round policy lookups (period / phase / timeout each read config) 24.83 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 4.82 microseconds
- concurrency 1, tier 10000 microseconds: rest of the scheduling round (due checks, locks, list upkeep) 1.43 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 40.83 microseconds; per-round policy lookups (period / phase / timeout each read config) 47.10 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 18.34 microseconds
- concurrency 4, tier 10000 microseconds: rest of the scheduling round (due checks, locks, list upkeep) 0.00 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 41.14 microseconds; per-round policy lookups (period / phase / timeout each read config) 44.81 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 8.34 microseconds

Attribution conclusions

- At concurrency 1 and a 40 microsecond body, the framework under test is not slower overall than `apscheduler` (-58 microseconds per task): the two sides have different timing structures, and the per-segment differences have both signs, so no 'which segment' is given.
- At concurrency 1 and a 40 microsecond body, the framework under test spends 88 microseconds more per task than `asyncio_pool`, most of it on 'submit side' (+47 microseconds).
- At concurrency 1 and a 40 microsecond body, the framework under test spends 85 microseconds more per task than `bare_thread`, most of it on 'handoff' (+119 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -79 microseconds, so read them together.
- At concurrency 1 and a 40 microsecond body, the framework under test spends 156 microseconds more per task than `thread_pool`, most of it on 'return' (+58 microseconds).
- In this group the framework's submit side consists of (microseconds): rest of the scheduling round (due checks, locks, list upkeep) 2, dispatch (handed to the scheduling model, incl. pool enqueue) 29, per-round policy lookups (period / phase / timeout each read config) 24, submit side, other (the adapter's own bookkeeping and completion-signal state) 10.
- At concurrency 4 and a 40 microsecond body, the framework under test is not slower overall than `apscheduler` (-247 microseconds per task): the two sides have different timing structures, and the per-segment differences have both signs, so no 'which segment' is given.

- At concurrency 4 and a 40 microsecond body, the framework under test spends 16 microseconds more per task than `asyncio_pool`, most of it on 'handoff' (+42 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'return' is -46 microseconds, so read them together.
- At concurrency 4 and a 40 microsecond body, the framework under test spends 200 microseconds more per task than `bare_thread`, most of it on 'handoff' (+379 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -97 microseconds, so read them together.
- At concurrency 4 and a 40 microsecond body, the framework under test spends 279 microseconds more per task than `thread_pool`, most of it on 'handoff' (+125 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'body' is -1 microseconds, so read them together.
- In this group the framework's submit side consists of (microseconds): rest of the scheduling round (due checks, locks, list upkeep) 0, dispatch (handed to the scheduling model, incl. pool enqueue) 24, per-round policy lookups (period / phase / timeout each read config) 25, submit side, other (the adapter's own bookkeeping and completion-signal state) 5.
- At concurrency 1 and a 10000 microsecond body, the framework under test is not slower overall than `apscheduler` (-184 microseconds per task): the two sides have different timing structures, and the per-segment differences have both signs, so no 'which segment' is given.
- At concurrency 1 and a 10000 microsecond body, the framework under test spends 325 microseconds more per task than `asyncio_pool`, most of it on 'body' (+217 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'handoff' is -1 microseconds, so read them together.
- At concurrency 1 and a 10000 microsecond body, the framework under test spends 121 microseconds more per task than `bare_thread`, most of it on 'handoff' (+5130 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -5092 microseconds, so read them together.
- At concurrency 1 and a 10000 microsecond body, the framework under test spends 401 microseconds more per task than `thread_pool`, most of it on 'body' (+213 microseconds).
- In this group the framework's submit side consists of (microseconds): rest of the scheduling round (due checks, locks, list upkeep) 1, dispatch (handed to the scheduling model, incl. pool enqueue) 41, per-round policy lookups (period / phase / timeout each read config) 47, submit side, other (the adapter's own bookkeeping and completion-signal state) 18.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 1529 microseconds more per task than `apscheduler`, most of it on 'body' (+5503 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'return' is -3941 microseconds, so read them together.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 1542 microseconds more per task than `asyncio_pool`, most of it on 'body' (+5767 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'return' is -4177 microseconds, so read them together.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 8639 microseconds more per task than `bare_thread`, most of it on 'handoff' (+15355 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -5141 microseconds, so read them together.

- At concurrency 4 and a 10000 microsecond body, the framework under test spends 2222 microseconds more per task than thread_pool, most of it on 'body' (+5396 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'return' is -3297 microseconds, so read them together.
- In this group the framework's submit side consists of (microseconds): rest of the scheduling round (due checks, locks, list upkeep) 0, dispatch (handed to the scheduling model, incl. pool enqueue) 41, per-round policy lookups (period / phase / timeout each read config) 45, submit side, other (the adapter's own bookkeeping and completion-signal state) 8.

Dimension: The cost of scheduling semantics

scope: this dimension uses an **internal switch comparison** (same workload, semantics all off vs. enabled one by one) and **does not compare across schemes** - a bare writeup has no counterpart for priority / timeout / retry, so it can only answer 'what is this framework's semantics worth', not 'how much more expensive than a bare writeup'

status: ok (the conclusion was reached on the 'current' drive-surface generation installed during this measurement)

semantics item	layer	how probed	measurable	evidence / reason
timeout	dispatch layer	behavioral probe (driven repeatedly through scheduling rounds; measures the moment the worker leaves the in-flight table)	yes	timeout reap observed: the worker left the in-flight table before the body ended (0.0122s) (body 0.15s, run_timeout 0.01s); judgment action at dispatch_core.py:472; what is in effect is observation and circuit-breaking (recording, marking unhealthy, removing the in-flight registration, stopping dispatch), not termination of the still-running worker - CPython cannot safely interrupt a running thread
priority	event layer	structural probe (checks whether the symbols required for execution exist)	no	not re-checked: this item's conclusion was reached on the 'previous' generation; taking it as this generation's fact asserts wiring in the report that the current generation does not have. Re-deriving it needs a probe design of its own (each item needs clean on/off states before it can be priced)

semantics item	layer	how probed	measurable	evidence / reason
retry	event layer	structural probe (checks whether the symbols required for execution exist)	no	not re-checked: this item's conclusion was reached on the 'previous' generation; taking it as this generation's fact asserts wiring in the report that the current generation does not have. Re-deriving it needs a probe design of its own (each item needs clean on/off states before it can be priced)

this conclusion was reached on the version under test and its coordinates hold for that version only; after appending a version to matrix.yaml, re-run this probe

Dimension: Native task line: three execution forms of the same subject

scope: this line is measured on its own: it enters neither the main matrix's pivot tables nor its crossings, and its numbers must not be read against the baseline adapters listed there

verdict: ok

equivalence gate: passed, 15 inputs compared byte for byte against the Python reference (envelope 256 bytes)

loaded task modbus_rtu.parse_response - Modbus RTU response-frame parsing: the input is a bare RTU ADU response frame (no transport framing), and the tier's frame bytes are its length

tiers R1, R8, R125 (frame bytes 7, 21, 255); concurrencies 1, 4, 16, 64; each cell runs 100 warm-up plus 100 measured rounds, one child process per concurrency (4 in all)

crossover (read at the judged concurrency 1 only): the native form is faster on R8, R125; slower on R1; undecided on none. crossover point: up to 7 frame bytes the native form is not faster, at or above 21 bytes it is

each figure is end-to-end per task (median of the measured rounds) - the same caliber as the main matrix's latency dimension

tier (microseconds)	frame bytes	concurrency	pure Python	native extension	online routing	native extension / pure Python	online routing / pure Python	faster form
R1	7	1	191.86 microseconds	197.11 microseconds	239.64 microseconds	1.03x	1.25x	pure Python
R1	7	4	265.75 microseconds	275.12 microseconds	277.12 microseconds	1.04x	1.04x	pure Python
R1	7	16	258.25 microseconds	270.49 microseconds	278.69 microseconds	1.05x	1.08x	pure Python
R1	7	64	253.67 microseconds	264.03 microseconds	281.22 microseconds	1.04x	1.11x	pure Python
R8	21	1	216.95 microseconds	205.05 microseconds	261.69 microseconds	0.95x	1.21x	native extension
R8	21	4	295.66 microseconds	289.60 microseconds	301.84 microseconds	0.98x	1.02x	undecided

tier (microseconds)	frame bytes	concurrency	pure Python	native extension	online routing	native extension / pure Python	online routing / pure Python	faster form
R8	21	16	289.49 microseconds	283.11 microseconds	306.98 microseconds	0.98x	1.06x	undecided
R8	21	64	277.33 microseconds	275.01 microseconds	306.46 microseconds	0.99x	1.11x	undecided
R125	255	1	642.48 microseconds	433.90 microseconds	487.97 microseconds	0.68x	0.76x	native extension
R125	255	4	708.90 microseconds	501.54 microseconds	514.68 microseconds	0.71x	0.73x	native extension
R125	255	16	692.00 microseconds	501.51 microseconds	513.39 microseconds	0.72x	0.74x	native extension
R125	255	64	670.87 microseconds	494.37 microseconds	518.43 microseconds	0.74x	0.77x	native extension

the routing verdict has two layers. Convergence: the arm actually taken is the majority of the arm counts inside the measured window, and the minority's share must be explainable by the exploration rate alone - counts are precise quantities, independent of wall time, and the per-round increments are archived so a reader can follow the settling. An arm the framework never took at all (cumulative snapshot, warm-up included) makes the majority the framework's default rather than a decision, and that is reported as such. Advantage: the two forms are contrasted by the paired per-task difference within one cycle (a sampled difference, not two separately measured medians subtracted), with no tolerance band (a band would turn 'inside it' into 'a tie'), and only when that difference's quartile range does not span zero - otherwise this caliber cannot tell them apart and the verdict is left undecided

routing at the judged concurrency, tier by tier (arm counts cover the measured window; the exploration rate is the framework's resolved value):

- R1 (7 bytes): arm counts native 1 / python 99; exploration rate 5.00% - converged to the python arm; agrees with the advantageous arm of the independent comparison (python arm)
- R8 (21 bytes): arm counts native 98 / python 2; exploration rate 5.00% - converged to the native arm; agrees with the advantageous arm of the independent comparison (native arm)
- R125 (255 bytes): arm counts native 98 / python 2; exploration rate 5.00% - converged to the native arm; agrees with the advantageous arm of the independent comparison (native arm)

caliber statements for this dimension:

- this dimension contrasts three execution forms of the same framework (pure Python / native extension / online routing); the figures come from the scheduling dispatch layer, the same level as the main matrix's zoo adapter. Only end-to-end per task is obtainable here - on this path the framework reports no per-unit duration, and this repository adds no instrumentation to it
- the dispatch layer's own overhead is a cost common to all three forms, so it compresses the ratios between them: the framework's own per-arm records (see the arm caliber note) show a much wider gap than end-to-end does. Read these ratios as 'running this task through the dispatch layer', not as the execution form's own speedup
- the crossover verdict is read at the lowest concurrency only: beyond the machine's parallel capacity the per-task wall time contains queueing wait, and then whether native is faster no longer measures the execution form (the other concurrency tiers are still measured and archived)

tiers where the framework under test is faster

Tiers where the framework under test is ahead (the two sides differ by more than that tier's band). This comes from the **same within-run comparison** as the openly published unfavourable data - one side takes the faster tiers, the other the slower ones, so both cover the same ground and neither can pick its own tiers. **This section being empty is not a failure**: it should be empty when nothing is distinguishable or when the framework loses everywhere.

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
apscheduler	1	40	the framework under test is 1.96x faster than apscheduler	157.03 microseconds
process_pool	1	40	the framework under test is 2.14x faster than process_pool	157.03 microseconds
apscheduler	1	300	the framework under test is 1.36x faster than apscheduler	412.58 microseconds
process_pool	1	300	the framework under test is 1.43x faster than process_pool	412.58 microseconds
apscheduler	1	2700	the framework under test is 1.06x faster than apscheduler	2.778 milliseconds
process_pool	1	2700	the framework under test is 1.08x faster than process_pool	2.778 milliseconds
apscheduler	4	40	the framework under test is 1.45x faster than apscheduler	177.06 microseconds
apscheduler	4	300	the framework under test is 1.24x faster than apscheduler	442.41 microseconds
apscheduler	4	2700	the framework under test is 1.04x faster than apscheduler	2.878 milliseconds
apscheduler	16	40	the framework under test is 1.45x faster than apscheduler	158.98 microseconds
apscheduler	16	300	the framework under test is 1.16x faster than apscheduler	440.91 microseconds
apscheduler	16	2700	the framework under test is 1.02x faster than apscheduler	2.885 milliseconds
apscheduler	64	40	the framework under test is 1.24x faster than apscheduler	194.80 microseconds
apscheduler	64	300	the framework under test is 1.13x faster than apscheduler	452.02 microseconds

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
apscheduler	64	2700	the framework under test is 1.02x faster than apscheduler	2.928 milliseconds

Openly published unfavourable data

Tiers where the framework under test is behind (the two sides differ by more than that tier's band). **A report missing this part is not acceptable** - a report that only shows where it wins gives the reader every reason to distrust it. Tiers whose difference is smaller than the band are not here but in the 'no winner' section: that is the verdict's outcome, not a place to hide things.

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
thread_pool	1	40	thread_pool is 1.98x faster than zoo	157.03 microseconds
thread_pool	1	300	thread_pool is 1.27x faster than zoo	412.58 microseconds
asyncio_pool	1	2700	asyncio_pool is 1.04x faster than zoo	2.778 milliseconds
thread_pool	1	2700	thread_pool is 1.06x faster than zoo	2.778 milliseconds
asyncio_pool	1	10000	asyncio_pool is 1.06x faster than zoo	9.929 milliseconds
bare_thread	1	10000	bare_thread is 1.03x faster than zoo	9.929 milliseconds
thread_pool	1	10000	thread_pool is 1.06x faster than zoo	9.929 milliseconds
thread_pool	4	40	thread_pool is 1.88x faster than zoo	177.06 microseconds
bare_thread	4	300	bare_thread is 1.07x faster than zoo	442.41 microseconds
process_pool	4	300	process_pool is 1.33x faster than zoo	442.41 microseconds
thread_pool	4	300	thread_pool is 1.27x faster than zoo	442.41 microseconds
asyncio_pool	4	2700	asyncio_pool is 1.05x faster than zoo	2.878 milliseconds
bare_thread	4	2700	bare_thread is 1.09x faster than zoo	2.878 milliseconds
process_pool	4	2700	process_pool is 2.06x faster than zoo	2.878 milliseconds
thread_pool	4	2700	thread_pool is 1.07x faster than zoo	2.878 milliseconds
bare_thread	4	10000	bare_thread is 1.03x faster than zoo	9.947 milliseconds
process_pool	4	10000	process_pool is 2.15x faster than zoo	9.947 milliseconds
asyncio_pool	16	40	asyncio_pool is 1.17x faster than zoo	158.98 microseconds

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
thread_pool	16	40	thread_pool is 1.76x faster than zoo	158.98 microseconds
asyncio_pool	16	300	asyncio_pool is 1.11x faster than zoo	440.91 microseconds
process_pool	16	300	process_pool is 1.59x faster than zoo	440.91 microseconds
thread_pool	16	300	thread_pool is 1.24x faster than zoo	440.91 microseconds
asyncio_pool	16	2700	asyncio_pool is 1.06x faster than zoo	2.885 milliseconds
bare_thread	16	2700	bare_thread is 1.07x faster than zoo	2.885 milliseconds
process_pool	16	2700	process_pool is 2.11x faster than zoo	2.885 milliseconds
thread_pool	16	2700	thread_pool is 1.05x faster than zoo	2.885 milliseconds
bare_thread	16	10000	bare_thread is 1.03x faster than zoo	9.988 milliseconds
process_pool	16	10000	process_pool is 2.13x faster than zoo	9.988 milliseconds
asyncio_pool	64	40	asyncio_pool is 1.57x faster than zoo	194.80 microseconds
bare_thread	64	40	bare_thread is 1.13x faster than zoo	194.80 microseconds
thread_pool	64	40	thread_pool is 2.15x faster than zoo	194.80 microseconds
asyncio_pool	64	300	asyncio_pool is 1.19x faster than zoo	452.02 microseconds
bare_thread	64	300	bare_thread is 1.06x faster than zoo	452.02 microseconds
process_pool	64	300	process_pool is 1.53x faster than zoo	452.02 microseconds
thread_pool	64	300	thread_pool is 1.32x faster than zoo	452.02 microseconds
asyncio_pool	64	2700	asyncio_pool is 1.05x faster than zoo	2.928 milliseconds
bare_thread	64	2700	bare_thread is 1.08x faster than zoo	2.928 milliseconds
process_pool	64	2700	process_pool is 2.16x faster than zoo	2.928 milliseconds
thread_pool	64	2700	thread_pool is 1.08x faster than zoo	2.928 milliseconds
bare_thread	64	10000	bare_thread is 1.08x faster than zoo	10.517 milliseconds
process_pool	64	10000	process_pool is 2.23x faster than zoo	10.517 milliseconds

the native-task line is a line of its own and is not mixed into the table above: at its judged concurrency 1 the native form is slower on R1 and the routing form is slower than pure interpretation on R1, R8; both are listed

with equal prominence, from the same paired comparison of the same run, in the "Native task line: three execution forms of the same subject" section

tiers with no winner

Tiers where the two sides **cannot be told apart**: the difference is smaller than that tier's band. The band comes from the two sides' own across-round spread (stdev / median of the per-task medians) **within the same run**, so it varies with the sample and the reader can recompute it from the archived per-round samples. It also covers every measured tier and states a reason for each - **this section is not a place to hide things**: reporting noise-level differences as '1.00x faster' is.

criterion: band = the framework's own across-round relative spread plus the baseline's; a single side's across-round relative spread = stdev / median of that unit's per-round per-task medians

band this run: 1.0% to 57.0% (it varies with each tier's spread).

baseline	concurrency	body tier (microseconds)	ratio	reason
asyncio_pool	1	40	0.92x	the two sides differ by 8.2%, less than the band of 23.0% - no winner
bare_thread	1	40	0.90x	the two sides differ by 9.8%, less than the band of 23.8% - no winner
asyncio_pool	1	300	0.94x	the two sides differ by 5.7%, less than the band of 7.2% - no winner
bare_thread	1	300	0.95x	the two sides differ by 4.6%, less than the band of 10.1% - no winner
bare_thread	1	2700	0.98x	the two sides differ by 1.9%, less than the band of 2.2% - no winner
apscheduler	1	10000	1.02x	the two sides differ by 1.8%, less than the band of 1.9% - no winner
process_pool	1	10000	1.02x	the two sides differ by 2.0%, less than the band of 3.7% - no winner
asyncio_pool	4	40	0.88x	the two sides differ by 11.6%, less than the band of 20.2% - no winner
bare_thread	4	40	0.92x	the two sides differ by 8.3%, less than the band of 17.7% - no winner

baseline	concurrency	body tier (microseconds)	ratio	reason
process_pool	4	40	1.00x	the two sides differ by 0.4%, less than the band of 30.2% - no winner
asyncio_pool	4	300	0.94x	the two sides differ by 6.2%, less than the band of 6.7% - no winner
apscheduler	4	10000	1.01x	the two sides differ by 1.5%, less than the band of 2.0% - no winner
asyncio_pool	4	10000	1.00x	the two sides differ by 0.1%, less than the band of 5.1% - no winner
thread_pool	4	10000	0.99x	the two sides differ by 1.1%, less than the band of 1.9% - no winner
bare_thread	16	40	1.06x	the two sides differ by 6.2%, less than the band of 7.3% - no winner
process_pool	16	40	1.00x	the two sides differ by 0.2%, less than the band of 21.0% - no winner
bare_thread	16	300	0.97x	the two sides differ by 3.0%, less than the band of 5.7% - no winner
apscheduler	16	10000	1.01x	the two sides differ by 1.1%, less than the band of 1.2% - no winner
asyncio_pool	16	10000	1.00x	the two sides differ by 0.1%, less than the band of 1.7% - no winner
thread_pool	16	10000	0.99x	the two sides differ by 1.2%, less than the band of 2.9% - no winner
process_pool	64	40	1.04x	the two sides differ by 3.9%, less than the band of 19.6% - no winner
apscheduler	64	10000	0.96x	the two sides differ by 3.7%, less than the band of 5.1% - no winner

baseline	concurrency	body tier (microseconds)	ratio	reason
asyncio_pool	64	10000	0.98x	the two sides differ by 2.5%, less than the band of 4.4% - no winner
thread_pool	64	10000	0.99x	the two sides differ by 1.0%, less than the band of 3.9% - no winner

caliber limitations and sources of bias

- **measured layer:** dispatch layer: drives BaseWaiter.execute_service() directly (thread-pool mode) instead of Master's full run loop - the latter's service loop fires once per second, which would lift the latency scale from microseconds to seconds. **One scheduling round per submitted task**, so that whole round's bookkeeping is charged to this single task; Master's usage serves several workers per round, which would spread that cost over tasks
- **scope note** (apscheduler): APScheduler's scheduled-job model, including job registration and trigger-decision cost; the library swallows exceptions raised inside jobs, so this tier re-raises them during drain. **Caliber note:** the library offers no job-completion notification, so the completion signal comes from this tier's Condition, and the cost of one notification per task counts toward its end-to-end
- **scope note** (asyncio_pool): the asyncio event loop dispatches **synchronous** bodies (equivalent to loop.run_in_executor), not native coroutine concurrency; the difference from a thread pool is the event loop's own cost
- **scope note** (bare_thread): one threading.Thread per task, no pooling and no reuse; thread creation cost counts toward end-to-end. Bodies' self-reported timings are collected with a lock-free list.append (atomic under the GIL) - **instrumentation must not charge one tier a cost other tiers do not pay**, or the comparison becomes a comparison of instrumentation
- **scope note** (process_pool): ProcessPoolExecutor: pickling the body and its return value across processes counts toward end-to-end, so this tier is not directly comparable to the multi-threaded tiers
- **scope note** (thread_pool): concurrent.futures.ThreadPoolExecutor, pool size = concurrency; threads are reused, with no task-queue semantics
- **scope note** (zoo): end-to-end stops at the worker completing and being deregistered from the in-flight table; the framework's later result delivery splits by generation: the previous generation's delivery path is broken (dispatch looks reactors up by reactor_name rather than by topic, so every call raises), and the completion signal can only come from the framework's internal callback; the current generation has that fixed, and the completion signal comes from the framework's own result reactor. **Caliber note:** the completion signal comes from this tier's Condition, and the cost of one notification per task counts toward its end-to-end; a bias in that direction makes the framework under test look slower, i.e. it works against it rather than for it
- **groups with no overhead figure:** for the (adapter, concurrency) groups below, concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values and meaningless change factors). This report therefore **gives them no framework overhead and no crossing point**; end-to-end, throughput and the body's self-reported values remain valid and not one field has been dropped from the raw archive. The criterion is the logical core count from the environment self-report, plus one evidential backstop: a computed negative overhead is likewise treated as unreadable - a framework can only add time, never remove it

- **self-check items excluded from the verdict:** for these units the body-tier deviation did not take part in the verdict (concurrency above the lowest, or a tier so short that wall clock is dominated by scheduling granularity). The raw values are still in place, but the wall clock at those two places is not enough to conclude whether the calibration is right - **factor this in when reading those tiers' numbers**

absolute durations: valid only within a single run; do not compare across runs

bring your own baseline

the baselines of this harness are **open**: implement the four methods of `zoo_bench.adapters.BaseAdapter` (setup / submit / drain / teardown), register with `zoo_bench.adapters.registry.register`, or list your module path in `extra_modules` of `matrix.yaml` - **no file in this repository needs to change**. Your implementation goes through the same equivalence verification (submit N bodies, assert all ran and each exactly once); failing it means the data is not trusted. **We deliberately write this path into the report**: a report written by the framework's own maintainer, measuring that framework, gives every reason to suspect a slowed baseline or a chosen caliber; letting anyone submit their own baseline is what makes this report challengeable and falsifiable.

self-check

- overall measurement self-check: passed
- process isolation: {'parent_pid': 71066, 'child_pids': [71522, 71525, 71528, 71531, 71534, 71540, 71546, 71552, 71558, 71576, 71594, 71612, 71630, 71687, 71745, 71800, 71866, 71868, 71870, 71872, 71874, 71879, 71884, 71890, 71895, 71912, 71929, 71946, 71963, 72025, 72087, 72152, 72221, 72255, 72289, 72323, 72357, 72490, 72623, 72756, 72889, 73418, 73947, 74476, 75006, 77119, 79232, 81345, 83460, 83464, 83468, 83472, 83476, 83483, 83490, 83497, 83504, 83523, 83542, 83561, 83580, 83647, 83714, 83781, 83852, 83854, 83856, 83858, 83860, 83865, 83870, 83875, 83880, 83897, 83914, 83931, 83948, 84004, 84061, 84120, 84187, 84189, 84191, 84193, 84195, 84200, 84205, 84210, 84215, 84232, 84251, 84268, 84285, 84350, 84415, 84480], 'all_distinct': True, 'none_equals_parent': True}

Appendix: all values

the body carries only pivot tables for readability; this appendix gives **every field as measured before any trimming** per unit (percentiles, spread, measured body durations, ...). **Not one value is dropped**: the details live inside the report body itself, not behind links, folds or attachments - 'verifiable' that requires a separate lookup is not verifiable

Appendix: Latency percentiles and jitter

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
apscheduler	1	40	308.03 microseconds	349.96 microseconds	358.97 microseconds	0.066
apscheduler	1	300	562.94 microseconds	604.91 microseconds	627.65 microseconds	0.046
apscheduler	1	2700	2.948 milliseconds	3.005 milliseconds	3.033 milliseconds	0.010
apscheduler	1	10000	10.112 milliseconds	10.352 milliseconds	10.407 milliseconds	0.012
apscheduler	4	40	257.17 microseconds	287.55 microseconds	303.15 microseconds	0.056

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
apscheduler	4	300	546.38 microseconds	561.11 microseconds	576.20 microseconds	0.022
apscheduler	4	2700	2.989 milliseconds	3.032 milliseconds	3.036 milliseconds	0.009
apscheduler	4	10000	10.095 milliseconds	10.212 milliseconds	10.494 milliseconds	0.011
apscheduler	16	40	230.76 microseconds	268.29 microseconds	271.09 microseconds	0.069
apscheduler	16	300	512.03 microseconds	538.09 microseconds	543.38 microseconds	0.024
apscheduler	16	2700	2.952 milliseconds	2.978 milliseconds	3.027 milliseconds	0.009
apscheduler	16	10000	10.097 milliseconds	10.192 milliseconds	10.227 milliseconds	0.006
apscheduler	64	40	241.67 microseconds	262.14 microseconds	262.49 microseconds	0.054
apscheduler	64	300	510.06 microseconds	530.38 microseconds	538.66 microseconds	0.027
apscheduler	64	2700	2.995 milliseconds	3.033 milliseconds	3.041 milliseconds	0.007
apscheduler	64	10000	10.128 milliseconds	10.588 milliseconds	10.631 milliseconds	0.024
asyncio_pool	1	40	144.14 microseconds	173.18 microseconds	187.49 microseconds	0.089
asyncio_pool	1	300	389.15 microseconds	418.60 microseconds	424.26 microseconds	0.036
asyncio_pool	1	2700	2.663 milliseconds	2.700 milliseconds	2.702 milliseconds	0.005
asyncio_pool	1	10000	9.399 milliseconds	9.941 milliseconds	10.816 milliseconds	0.036
asyncio_pool	4	40	156.44 microseconds	177.87 microseconds	184.77 microseconds	0.080
asyncio_pool	4	300	414.86 microseconds	432.37 microseconds	438.97 microseconds	0.023
asyncio_pool	4	2700	2.729 milliseconds	2.771 milliseconds	2.784 milliseconds	0.009
asyncio_pool	4	10000	9.955 milliseconds	10.372 milliseconds	11.706 milliseconds	0.042
asyncio_pool	16	40	136.04 microseconds	147.86 microseconds	158.86 microseconds	0.061
asyncio_pool	16	300	395.94 microseconds	412.78 microseconds	419.97 microseconds	0.024
asyncio_pool	16	2700	2.725 milliseconds	2.760 milliseconds	2.766 milliseconds	0.009
asyncio_pool	16	10000	9.980 milliseconds	10.087 milliseconds	10.399 milliseconds	0.012
asyncio_pool	64	40	124.21 microseconds	127.64 microseconds	128.05 microseconds	0.018

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
asyncio_pool	64	300	378.92 microseconds	383.97 microseconds	386.20 microseconds	0.008
asyncio_pool	64	2700	2.779 milliseconds	2.808 milliseconds	2.811 milliseconds	0.008
asyncio_pool	64	10000	10.259 milliseconds	10.474 milliseconds	10.528 milliseconds	0.018
bare_thread	1	40	141.61 microseconds	176.49 microseconds	185.39 microseconds	0.097
bare_thread	1	300	393.51 microseconds	447.16 microseconds	485.24 microseconds	0.066
bare_thread	1	2700	2.725 milliseconds	2.778 milliseconds	2.783 milliseconds	0.012
bare_thread	1	10000	9.630 milliseconds	9.751 milliseconds	9.889 milliseconds	0.008
bare_thread	4	40	162.35 microseconds	181.13 microseconds	187.59 microseconds	0.055
bare_thread	4	300	411.77 microseconds	429.43 microseconds	433.87 microseconds	0.023
bare_thread	4	2700	2.648 milliseconds	2.675 milliseconds	2.680 milliseconds	0.005
bare_thread	4	10000	9.618 milliseconds	9.680 milliseconds	9.697 milliseconds	0.004
bare_thread	16	40	168.76 microseconds	176.90 microseconds	178.35 microseconds	0.023
bare_thread	16	300	427.79 microseconds	433.37 microseconds	434.78 microseconds	0.007
bare_thread	16	2700	2.700 milliseconds	2.718 milliseconds	2.727 milliseconds	0.004
bare_thread	16	10000	9.717 milliseconds	9.795 milliseconds	9.860 milliseconds	0.004
bare_thread	64	40	171.85 microseconds	175.73 microseconds	176.05 microseconds	0.014
bare_thread	64	300	427.59 microseconds	431.98 microseconds	433.04 microseconds	0.006
bare_thread	64	2700	2.705 milliseconds	2.724 milliseconds	2.728 milliseconds	0.004
bare_thread	64	10000	9.718 milliseconds	9.769 milliseconds	9.779 milliseconds	0.004
process_pool	1	40	336.29 microseconds	371.95 microseconds	907.07 microseconds	0.429
process_pool	1	300	591.15 microseconds	634.03 microseconds	1.158 milliseconds	0.246
process_pool	1	2700	3.013 milliseconds	3.061 milliseconds	3.668 milliseconds	0.055
process_pool	1	10000	10.125 milliseconds	11.006 milliseconds	11.208 milliseconds	0.030
process_pool	4	40	176.27 microseconds	244.14 microseconds	252.09 microseconds	0.181

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
process_pool	4	300	331.48 microseconds	369.10 microseconds	393.42 microseconds	0.094
process_pool	4	2700	1.400 milliseconds	1.518 milliseconds	1.594 milliseconds	0.041
process_pool	4	10000	4.617 milliseconds	5.118 milliseconds	5.600 milliseconds	0.053
process_pool	16	40	158.72 microseconds	204.66 microseconds	211.37 microseconds	0.160
process_pool	16	300	276.84 microseconds	315.26 microseconds	322.45 microseconds	0.079
process_pool	16	2700	1.370 milliseconds	1.469 milliseconds	1.516 milliseconds	0.032
process_pool	16	10000	4.698 milliseconds	4.911 milliseconds	4.934 milliseconds	0.019
process_pool	64	40	202.33 microseconds	237.75 microseconds	243.81 microseconds	0.102
process_pool	64	300	296.22 microseconds	334.60 microseconds	342.73 microseconds	0.058
process_pool	64	2700	1.357 milliseconds	1.410 milliseconds	1.445 milliseconds	0.019
process_pool	64	10000	4.709 milliseconds	4.740 milliseconds	4.866 milliseconds	0.009
thread_pool	1	40	79.31 microseconds	101.71 microseconds	132.80 microseconds	0.184
thread_pool	1	300	325.37 microseconds	362.51 microseconds	375.66 microseconds	0.053
thread_pool	1	2700	2.623 milliseconds	2.711 milliseconds	2.937 milliseconds	0.029
thread_pool	1	10000	9.369 milliseconds	9.476 milliseconds	9.510 milliseconds	0.007
thread_pool	4	40	94.18 microseconds	107.91 microseconds	109.81 microseconds	0.076
thread_pool	4	300	348.68 microseconds	363.07 microseconds	364.89 microseconds	0.019
thread_pool	4	2700	2.702 milliseconds	2.740 milliseconds	2.761 milliseconds	0.011
thread_pool	4	10000	9.839 milliseconds	10.035 milliseconds	10.105 milliseconds	0.010
thread_pool	16	40	90.57 microseconds	99.86 microseconds	104.29 microseconds	0.057
thread_pool	16	300	355.80 microseconds	370.72 microseconds	376.40 microseconds	0.019
thread_pool	16	2700	2.742 milliseconds	2.786 milliseconds	2.800 milliseconds	0.009
thread_pool	16	10000	9.867 milliseconds	10.048 milliseconds	10.787 milliseconds	0.023
thread_pool	64	40	90.47 microseconds	92.30 microseconds	92.47 microseconds	0.018

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
thread_pool	64	300	341.71 microseconds	347.75 microseconds	348.64 microseconds	0.009
thread_pool	64	2700	2.705 milliseconds	2.798 milliseconds	2.946 milliseconds	0.023
thread_pool	64	10000	10.415 milliseconds	10.509 milliseconds	10.544 milliseconds	0.012
zoo	1	40	157.03 microseconds	210.32 microseconds	237.86 microseconds	0.141
zoo	1	300	412.58 microseconds	445.22 microseconds	451.32 microseconds	0.036
zoo	1	2700	2.778 milliseconds	2.829 milliseconds	2.873 milliseconds	0.010
zoo	1	10000	9.929 milliseconds	10.006 milliseconds	10.039 milliseconds	0.007
zoo	4	40	177.06 microseconds	230.68 microseconds	233.14 microseconds	0.122
zoo	4	300	442.41 microseconds	486.15 microseconds	486.91 microseconds	0.044
zoo	4	2700	2.878 milliseconds	2.957 milliseconds	2.975 milliseconds	0.017
zoo	4	10000	9.947 milliseconds	10.070 milliseconds	10.087 milliseconds	0.009
zoo	16	40	158.98 microseconds	172.71 microseconds	180.55 microseconds	0.050
zoo	16	300	440.91 microseconds	485.64 microseconds	488.09 microseconds	0.050
zoo	16	2700	2.885 milliseconds	2.933 milliseconds	2.957 milliseconds	0.011
zoo	16	10000	9.988 milliseconds	10.064 milliseconds	10.066 milliseconds	0.006
zoo	64	40	194.80 microseconds	218.06 microseconds	219.26 microseconds	0.095
zoo	64	300	452.02 microseconds	463.04 microseconds	471.99 microseconds	0.022
zoo	64	2700	2.928 milliseconds	3.004 milliseconds	3.061 milliseconds	0.015
zoo	64	10000	10.517 milliseconds	10.925 milliseconds	11.084 milliseconds	0.027

Appendix: Framework overhead share

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
apscheduler	1	40	250.47 microseconds	81.31%	57.56 microseconds
apscheduler	1	300	261.46 microseconds	46.45%	301.48 microseconds
apscheduler	1	2700	277.96 microseconds	9.43%	2.670 milliseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
apscheduler	1	10000	442.08 microseconds	4.37%	9.670 milliseconds
apscheduler	4	40	198.84 microseconds	77.32%	58.33 microseconds
apscheduler	4	300	234.40 microseconds	42.90%	311.98 microseconds
apscheduler	4	2700	307.46 microseconds	10.29%	2.682 milliseconds
apscheduler	4	10000	-	-	14.587 milliseconds
apscheduler	16	40	-	-	53.43 microseconds
apscheduler	16	300	-	-	312.27 microseconds
apscheduler	16	2700	-	-	2.685 milliseconds
apscheduler	16	10000	-	-	18.654 milliseconds
apscheduler	64	40	-	-	51.87 microseconds
apscheduler	64	300	-	-	295.36 microseconds
apscheduler	64	2700	-	-	2.740 milliseconds
apscheduler	64	10000	-	-	19.471 milliseconds
asyncio_pool	1	40	97.94 microseconds	67.95%	46.20 microseconds
asyncio_pool	1	300	104.53 microseconds	26.86%	284.63 microseconds
asyncio_pool	1	2700	129.72 microseconds	4.87%	2.534 milliseconds
asyncio_pool	1	10000	140.82 microseconds	1.50%	9.258 milliseconds
asyncio_pool	4	40	104.52 microseconds	66.81%	51.92 microseconds
asyncio_pool	4	300	106.23 microseconds	25.61%	308.64 microseconds
asyncio_pool	4	2700	152.12 microseconds	5.57%	2.577 milliseconds
asyncio_pool	4	10000	-	-	14.730 milliseconds
asyncio_pool	16	40	-	-	53.92 microseconds
asyncio_pool	16	300	-	-	304.81 microseconds
asyncio_pool	16	2700	-	-	2.605 milliseconds
asyncio_pool	16	10000	-	-	20.437 milliseconds
asyncio_pool	64	40	-	-	52.81 microseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
asyncio_pool	64	300	-	-	299.54 microseconds
asyncio_pool	64	2700	-	-	2.649 milliseconds
asyncio_pool	64	10000	-	-	59.077 milliseconds
bare_thread	1	40	89.53 microseconds	63.22%	52.09 microseconds
bare_thread	1	300	97.34 microseconds	24.74%	296.17 microseconds
bare_thread	1	2700	116.72 microseconds	4.28%	2.608 milliseconds
bare_thread	1	10000	115.64 microseconds	1.20%	9.514 milliseconds
bare_thread	4	40	110.79 microseconds	68.24%	51.56 microseconds
bare_thread	4	300	119.68 microseconds	29.06%	292.09 microseconds
bare_thread	4	2700	115.99 microseconds	4.38%	2.532 milliseconds
bare_thread	4	10000	-	-	14.615 milliseconds
bare_thread	16	40	-	-	50.83 microseconds
bare_thread	16	300	-	-	294.67 microseconds
bare_thread	16	2700	-	-	2.563 milliseconds
bare_thread	16	10000	-	-	14.718 milliseconds
bare_thread	64	40	-	-	50.76 microseconds
bare_thread	64	300	-	-	295.36 microseconds
bare_thread	64	2700	-	-	2.565 milliseconds
bare_thread	64	10000	-	-	14.787 milliseconds
process_pool	1	40	281.77 microseconds	83.79%	54.52 microseconds
process_pool	1	300	282.90 microseconds	47.86%	308.25 microseconds
process_pool	1	2700	353.01 microseconds	11.72%	2.660 milliseconds
process_pool	1	10000	370.20 microseconds	3.66%	9.755 milliseconds
process_pool	4	40	81.93 microseconds	46.48%	94.34 microseconds
process_pool	4	300	-	-	571.62 microseconds
process_pool	4	2700	-	-	4.823 milliseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
process_pool	4	10000	-	-	17.617 milliseconds
process_pool	16	40	-	-	90.76 microseconds
process_pool	16	300	-	-	588.78 microseconds
process_pool	16	2700	-	-	9.835 milliseconds
process_pool	16	10000	-	-	52.389 milliseconds
process_pool	64	40	-	-	89.98 microseconds
process_pool	64	300	-	-	575.97 microseconds
process_pool	64	2700	-	-	27.524 milliseconds
process_pool	64	10000	-	-	167.164 milliseconds
thread_pool	1	40	31.15 microseconds	39.28%	48.16 microseconds
thread_pool	1	300	37.71 microseconds	11.59%	287.66 microseconds
thread_pool	1	2700	51.42 microseconds	1.96%	2.572 milliseconds
thread_pool	1	10000	56.20 microseconds	0.60%	9.313 milliseconds
thread_pool	4	40	46.09 microseconds	48.94%	48.09 microseconds
thread_pool	4	300	53.22 microseconds	15.26%	295.45 microseconds
thread_pool	4	2700	109.72 microseconds	4.06%	2.592 milliseconds
thread_pool	4	10000	-	-	19.244 milliseconds
thread_pool	16	40	-	-	48.03 microseconds
thread_pool	16	300	-	-	305.97 microseconds
thread_pool	16	2700	-	-	2.660 milliseconds
thread_pool	16	10000	-	-	20.155 milliseconds
thread_pool	64	40	-	-	47.00 microseconds
thread_pool	64	300	-	-	285.50 microseconds
thread_pool	64	2700	-	-	2.624 milliseconds
thread_pool	64	10000	-	-	76.944 milliseconds
zoo	1	40	107.81 microseconds	68.65%	49.22 microseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
zoo	1	300	114.25 microseconds	27.69%	298.32 microseconds
zoo	1	2700	139.00 microseconds	5.00%	2.639 milliseconds
zoo	1	10000	173.17 microseconds	1.74%	9.756 milliseconds
zoo	4	40	128.15 microseconds	72.38%	48.91 microseconds
zoo	4	300	139.93 microseconds	31.63%	302.48 microseconds
zoo	4	2700	155.55 microseconds	5.40%	2.723 milliseconds
zoo	4	10000	-	-	14.996 milliseconds
zoo	16	40	-	-	42.93 microseconds
zoo	16	300	-	-	302.36 microseconds
zoo	16	2700	-	-	2.737 milliseconds
zoo	16	10000	-	-	25.176 milliseconds
zoo	64	40	-	-	47.76 microseconds
zoo	64	300	-	-	284.11 microseconds
zoo	64	2700	-	-	2.747 milliseconds
zoo	64	10000	-	-	114.364 milliseconds

Appendix: Throughput and concurrency scaling

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
apscheduler	1	40	3246.5
apscheduler	1	300	1776.4
apscheduler	1	2700	339.2
apscheduler	1	10000	98.9
apscheduler	4	40	15554.0
apscheduler	4	300	7321.0
apscheduler	4	2700	1338.2
apscheduler	4	10000	396.3
apscheduler	16	40	69336.7
apscheduler	16	300	31248.4
apscheduler	16	2700	5419.6
apscheduler	16	10000	1584.6
apscheduler	64	40	264822.7
apscheduler	64	300	125476.2
apscheduler	64	2700	21372.5

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
apscheduler	64	10000	6319.2
asyncio_pool	1	40	6937.9
asyncio_pool	1	300	2569.7
asyncio_pool	1	2700	375.5
asyncio_pool	1	10000	106.4
asyncio_pool	4	40	25569.1
asyncio_pool	4	300	9641.7
asyncio_pool	4	2700	1465.6
asyncio_pool	4	10000	401.8
asyncio_pool	16	40	117614.4
asyncio_pool	16	300	40410.1
asyncio_pool	16	2700	5872.3
asyncio_pool	16	10000	1603.2
asyncio_pool	64	40	515270.2
asyncio_pool	64	300	168900.1
asyncio_pool	64	2700	23028.2
asyncio_pool	64	10000	6238.4
bare_thread	1	40	7061.4
bare_thread	1	300	2541.2
bare_thread	1	2700	367.0
bare_thread	1	10000	103.8
bare_thread	4	40	24638.4
bare_thread	4	300	9714.1
bare_thread	4	2700	1510.8
bare_thread	4	10000	415.9
bare_thread	16	40	94809.1
bare_thread	16	300	37401.2
bare_thread	16	2700	5926.5
bare_thread	16	10000	1646.6
bare_thread	64	40	372428.2
bare_thread	64	300	149676.6
bare_thread	64	2700	23658.9
bare_thread	64	10000	6585.6
process_pool	1	40	2973.7
process_pool	1	300	1691.6
process_pool	1	2700	331.9
process_pool	1	10000	98.8
process_pool	4	40	22692.6
process_pool	4	300	12067.1
process_pool	4	2700	2856.9
process_pool	4	10000	866.4
process_pool	16	40	100804.1
process_pool	16	300	57794.7
process_pool	16	2700	11681.1

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
process_pool	16	10000	3405.5
process_pool	64	40	316315.6
process_pool	64	300	216054.3
process_pool	64	2700	47176.6
process_pool	64	10000	13590.5
thread_pool	1	40	12609.0
thread_pool	1	300	3073.4
thread_pool	1	2700	381.2
thread_pool	1	10000	106.7
thread_pool	4	40	42474.1
thread_pool	4	300	11472.0
thread_pool	4	2700	1480.5
thread_pool	4	10000	406.6
thread_pool	16	40	176656.6
thread_pool	16	300	44969.3
thread_pool	16	2700	5835.0
thread_pool	16	10000	1621.6
thread_pool	64	40	707442.1
thread_pool	64	300	187293.3
thread_pool	64	2700	23661.8
thread_pool	64	10000	6144.8
zoo	1	40	6368.3
zoo	1	300	2423.8
zoo	1	2700	359.9
zoo	1	10000	100.7
zoo	4	40	22591.2
zoo	4	300	9041.4
zoo	4	2700	1389.7
zoo	4	10000	402.1
zoo	16	40	100640.5
zoo	16	300	36288.7
zoo	16	2700	5545.8
zoo	16	10000	1601.9
zoo	64	40	328542.1
zoo	64	300	141586.6
zoo	64	2700	21859.0
zoo	64	10000	6085.3