

zoo-bench performance report

raw data: results/zoo-framework-dev/20261011T094251Z.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)	0.11.2-beta
framework under test (module __version__)	0.11.2-beta (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)
install source	dev via git
commit	44dcbca5f392
harness	zoo-bench 0.0.1 @ 9a42be24ac43f75e2e5d7b16434f91ba3d0717b6
command to reproduce	zoo-bench run --framework zoo-framework @ git+https://github.com/YearsAlso/zoo-framework.git@dev

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 2.31x 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, 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 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, 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 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, 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 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	-	not observed: within the measured tiers this baseline is always faster than the framework under test
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	-	not observed: within the measured tiers this baseline is always faster than the framework under test
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	-	not observed: within the measured tiers this baseline is always faster than the framework under test
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	151.05 microseconds	287.86 microseconds	142.30 microseconds	145.22 microseconds	349.41 microseconds	75.73 microseconds
1	300	410.28 microseconds	553.65 microseconds	385.12 microseconds	388.13 microseconds	590.88 microseconds	313.96 microseconds
1	2700	2.753 milliseconds	2.987 milliseconds	2.699 milliseconds	2.634 milliseconds	2.988 milliseconds	2.667 milliseconds

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
1	10000	9.871 milliseconds	9.941 milliseconds	9.474 milliseconds	9.553 milliseconds	9.773 milliseconds	9.525 milliseconds
4	40	174.59 microseconds	252.70 microseconds	162.47 microseconds	158.14 microseconds	167.32 microseconds	97.54 microseconds
4	300	449.42 microseconds	521.58 microseconds	422.43 microseconds	412.54 microseconds	306.62 microseconds	346.42 microseconds
4	2700	2.911 milliseconds	2.912 milliseconds	2.754 milliseconds	2.683 milliseconds	1.390 milliseconds	2.689 milliseconds
4	10000	9.949 milliseconds	9.993 milliseconds	9.670 milliseconds	9.713 milliseconds	4.653 milliseconds	9.529 milliseconds
16	40	177.55 microseconds	229.36 microseconds	133.81 microseconds	171.28 microseconds	156.46 microseconds	90.20 microseconds
16	300	466.12 microseconds	505.80 microseconds	396.45 microseconds	423.99 microseconds	275.04 microseconds	345.34 microseconds
16	2700	2.908 milliseconds	2.947 milliseconds	2.770 milliseconds	2.705 milliseconds	1.381 milliseconds	2.740 milliseconds
16	10000	10.146 milliseconds	10.216 milliseconds	9.917 milliseconds	9.716 milliseconds	4.712 milliseconds	9.785 milliseconds
64	40	205.09 microseconds	221.11 microseconds	121.88 microseconds	170.85 microseconds	204.14 microseconds	91.39 microseconds
64	300	462.64 microseconds	509.59 microseconds	381.72 microseconds	425.01 microseconds	307.25 microseconds	352.74 microseconds
64	2700	2.884 milliseconds	2.905 milliseconds	2.779 milliseconds	2.746 milliseconds	1.379 milliseconds	2.647 milliseconds
64	10000	10.447 milliseconds	10.197 milliseconds	9.811 milliseconds	9.532 milliseconds	4.713 milliseconds	10.238 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	67.93%	82.43%	67.30%	64.62%	84.22%	41.18%
1	300	28.03%	45.95%	27.77%	26.48%	49.33%	11.25%
1	2700	5.00%	9.52%	4.39%	4.23%	11.32%	1.80%
1	10000	1.63%	3.50%	1.71%	1.36%	3.65%	0.60%
4	40	72.39%	77.98%	67.61%	68.53%	43.29%	48.41%
4	300	33.16%	40.01%	26.84%	27.97%	-	15.15%
4	2700	6.07%	9.25%	5.48%	4.89%	-	3.51%
4	10000	-	-	-	-	-	-
16	40	-	-	-	-	-	-
16	300	-	-	-	-	-	-
16	2700	-	-	-	-	-	-

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
16	10000	-	-	-	-	-	-
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	6620.4	3473.9	7027.6	6886.0	2862.0	13204.5
1	300	2437.4	1806.2	2596.6	2576.4	1692.4	3185.1
1	2700	363.2	334.8	370.4	379.6	334.6	375.0
1	10000	101.3	100.6	105.5	104.7	102.3	105.0
4	40	22910.9	15828.8	24620.6	25293.9	23906.0	41009.0
4	300	8900.3	7669.0	9468.9	9696.1	13045.3	11546.7
4	2700	1374.2	1373.4	1452.5	1491.0	2878.5	1487.4
4	10000	402.1	400.3	413.6	411.8	859.6	419.8
16	40	90116.9	69759.2	119571.2	93414.7	102262.4	177375.4
16	300	34326.2	31632.8	40357.7	37737.0	58172.4	46331.7
16	2700	5501.6	5429.2	5775.3	5915.8	11588.2	5838.4
16	10000	1577.0	1566.2	1613.4	1646.8	3395.6	1635.2
64	40	312054.2	289446.9	525124.2	374606.5	313503.0	700311.8
64	300	138336.3	125590.4	167662.6	150584.2	208301.1	181435.9
64	2700	22191.3	22031.0	23032.2	23310.3	46410.5	24173.8
64	10000	6126.4	6276.1	6523.5	6714.2	13578.0	6251.0

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	108.63 microseconds	91.69 microseconds	50.72 microseconds	40.59 microseconds	ok
apscheduler	4	40	109.96 microseconds	509.17 microseconds	53.33 microseconds	229.15 microseconds	ok
apscheduler	1	10000	165.03 microseconds	115.08 microseconds	9.620 milliseconds	54.23 microseconds	ok
apscheduler	4	10000	124.65 microseconds	10.534 milliseconds	14.747 milliseconds	9.563 milliseconds	ok
asyncio_pool	1	40	17.86 microseconds	27.78 microseconds	46.74 microseconds	61.41 microseconds	ok
asyncio_pool	4	40	19.51 microseconds	277.63 microseconds	56.04 microseconds	298.30 microseconds	ok
asyncio_pool	1	10000	24.40 microseconds	38.13 microseconds	9.282 milliseconds	88.21 microseconds	ok
asyncio_pool	4	10000	23.24 microseconds	12.312 milliseconds	14.438 milliseconds	9.637 milliseconds	ok
bare_thread	1	40	142.97 microseconds	-77.02 microseconds	53.08 microseconds	31.00 microseconds	ok
bare_thread	4	40	156.85 microseconds	-76.12 microseconds	51.67 microseconds	320.79 microseconds	ok
bare_thread	1	10000	5.183 milliseconds	-5103.28 microseconds	9.413 milliseconds	40.28 microseconds	ok
bare_thread	4	10000	5.193 milliseconds	-5086.31 microseconds	14.456 milliseconds	9.546 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	8.43 microseconds	10.97 microseconds	47.27 microseconds	15.89 microseconds	ok
thread_pool	4	40	11.32 microseconds	189.17 microseconds	51.76 microseconds	162.73 microseconds	ok
thread_pool	1	10000	12.30 microseconds	17.07 microseconds	9.423 milliseconds	27.84 microseconds	ok
thread_pool	4	10000	13.63 microseconds	10.044 milliseconds	14.793 milliseconds	9.433 milliseconds	ok
zoo	1	40	51.28 microseconds	24.88 microseconds	53.98 microseconds	57.64 microseconds	ok
zoo	4	40	61.92 microseconds	303.25 microseconds	47.79 microseconds	269.05 microseconds	ok

scheme	concurrency	tier (microse conds)	submit side	handoff	body	return	status
zoo	1	10000	94.33 microseconds	37.15 microseconds	9.568 milliseconds	120.28 microseconds	ok
zoo	4	10000	94.83 microseconds	10.189 milliseconds	15.075 milliseconds	9.820 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	-107.03 microseconds	return	17.05 microseconds
asyncio_pool	1	40	34.72 microseconds	submit side	33.42 microseconds
bare_thread	1	40	39.11 microseconds	handoff	101.91 microseconds
thread_pool	1	40	106.77 microseconds	submit side	42.86 microseconds
apscheduler	4	40	-249.64 microseconds	return	39.89 microseconds
asyncio_pool	4	40	4.87 microseconds	submit side	42.41 microseconds
bare_thread	4	40	227.67 microseconds	handoff	379.38 microseconds
thread_pool	4	40	266.49 microseconds	handoff	114.08 microseconds
apscheduler	1	10000	-125.56 microseconds	return	66.05 microseconds
asyncio_pool	1	10000	370.28 microseconds	body	285.17 microseconds
bare_thread	1	10000	272.87 microseconds	handoff	5.140 milliseconds
thread_pool	1	10000	337.26 microseconds	body	144.70 microseconds
apscheduler	4	10000	102.47 microseconds	body	328.30 microseconds
asyncio_pool	4	10000	1.119 milliseconds	body	637.18 microseconds
bare_thread	4	10000	9.802 milliseconds	handoff	15.276 milliseconds
thread_pool	4	10000	978.60 microseconds	return	387.43 microseconds

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

184.05 microseconds vs with 189.35 microseconds, a difference of 5.30 microseconds; zoo 40 microseconds /

concurrency 4: without instrumentation 702.46 microseconds vs with 678.24 microseconds, a difference of 24.23 microseconds (faster with instrumentation: this round's machine drift exceeded the instrumentation cost); zoo 10000 microseconds / concurrency 1: without instrumentation 9.878 milliseconds vs with 9.827 milliseconds, a difference of 50.79 microseconds (faster with instrumentation: this round's machine drift exceeded the instrumentation cost); zoo 10000 microseconds / concurrency 4: without instrumentation 40.260 milliseconds vs with 40.088 milliseconds, a difference of 171.69 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 5.9%; zoo 10000 microseconds / concurrency 4 overlaps 92.1%

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) 0.38 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 21.95 microseconds; per-round policy lookups (period / phase / timeout each read config) 20.87 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 8.23 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) 30.72 microseconds; per-round policy lookups (period / phase / timeout each read config) 29.09 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 4.37 microseconds
- concurrency 1, tier 10000 microseconds: rest of the scheduling round (due checks, locks, list upkeep) 0.92 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 37.69 microseconds; per-round policy lookups (period / phase / timeout each read config) 38.45 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 16.14 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) 42.46 microseconds; per-round policy lookups (period / phase / timeout each read config) 50.79 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 7.84 microseconds

Attribution conclusions

- At concurrency 1 and a 40 microsecond body, the framework under test is not slower overall than apscheduler (-107 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 35 microseconds more per task than asyncio_pool, most of it on 'submit side' (+33 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'return' is -4 microseconds, so read them together.
- At concurrency 1 and a 40 microsecond body, the framework under test spends 39 microseconds more per task than bare_thread, most of it on 'handoff' (+102 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -92 microseconds, so read them together.
- At concurrency 1 and a 40 microsecond body, the framework under test spends 107 microseconds more per task than thread_pool, most of it on 'submit side' (+43 microseconds).
- 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) 22, per-round

policy lookups (period / phase / timeout each read config) 21, submit side, other (the adapter's own bookkeeping and completion-signal state) 8.

- At concurrency 4 and a 40 microsecond body, the framework under test is not slower overall than `apscheduler` (-250 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 5 microseconds more per task than `asyncio_pool`, most of it on 'submit side' (+42 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'return' is -29 microseconds, so read them together.
- At concurrency 4 and a 40 microsecond body, the framework under test spends 228 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 -95 microseconds, so read them together.
- At concurrency 4 and a 40 microsecond body, the framework under test spends 266 microseconds more per task than `thread_pool`, most of it on 'handoff' (+114 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'body' is -4 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) 31, per-round policy lookups (period / phase / timeout each read config) 29, submit side, other (the adapter's own bookkeeping and completion-signal state) 4.
- At concurrency 1 and a 10000 microsecond body, the framework under test is not slower overall than `apscheduler` (-126 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 370 microseconds more per task than `asyncio_pool`, most of it on 'body' (+285 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 273 microseconds more per task than `bare_thread`, most of it on 'handoff' (+5140 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -5089 microseconds, so read them together.
- At concurrency 1 and a 10000 microsecond body, the framework under test spends 337 microseconds more per task than `thread_pool`, most of it on 'body' (+145 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) 38, per-round policy lookups (period / phase / timeout each read config) 38, submit side, other (the adapter's own bookkeeping and completion-signal state) 16.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 102 microseconds more per task than `apscheduler`, most of it on 'body' (+328 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'handoff' is -345 microseconds, so read them together.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 1119 microseconds more per task than `asyncio_pool`, most of it on 'body' (+637 microseconds).

- note: the per-segment differences in this group do not all share a sign - 'handoff' is -2123 microseconds, so read them together.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 9802 microseconds more per task than bare_thread, most of it on 'handoff' (+15276 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -5098 microseconds, so read them together.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 979 microseconds more per task than thread_pool, most of it on 'return' (+387 microseconds).
- 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) 42, per-round policy lookups (period / phase / timeout each read config) 51, 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

semantics item	layer	how probed	measurable	evidence / reason
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)
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	189.78 microseconds	192.82 microseconds	239.17 microseconds	1.02x	1.26x	pure Python
R1	7	4	260.82 microseconds	268.30 microseconds	277.48 microseconds	1.03x	1.06x	pure Python
R1	7	16	255.22 microseconds	264.64 microseconds	275.81 microseconds	1.04x	1.08x	pure Python
R1	7	64	257.22 microseconds	266.34 microseconds	282.83 microseconds	1.04x	1.10x	undecided
R8	21	1	216.56 microseconds	205.28 microseconds	258.46 microseconds	0.95x	1.19x	native extension
R8	21	4	291.65 microseconds	271.99 microseconds	304.52 microseconds	0.93x	1.04x	undecided
R8	21	16	284.36 microseconds	278.61 microseconds	303.14 microseconds	0.98x	1.07x	undecided
R8	21	64	282.44 microseconds	281.62 microseconds	307.27 microseconds	1.00x	1.09x	undecided
R125	255	1	643.52 microseconds	410.37 microseconds	470.20 microseconds	0.64x	0.73x	native extension
R125	255	4	698.27 microseconds	495.83 microseconds	515.80 microseconds	0.71x	0.74x	native extension
R125	255	16	688.63 microseconds	498.11 microseconds	516.04 microseconds	0.72x	0.75x	native extension
R125	255	64	675.87 microseconds	497.24 microseconds	525.11 microseconds	0.74x	0.78x	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 100 / python 0; exploration rate 5.00% - size R125: the python arm was never taken in this group's 100 decisions, so the arm that took the majority is the framework's default (used when the two means are equal or unmeasured) rather than a decision this run reached. The independent comparison still yields a verdict; this note says how much it rests on; 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 1.91x faster than aperiodic	151.05 microseconds
process_pool	1	40	the framework under test is 2.31x faster than process_pool	151.05 microseconds
apscheduler	1	300	the framework under test is 1.35x faster than aperiodic	410.28 microseconds
process_pool	1	300	the framework under test is 1.44x faster than process_pool	410.28 microseconds
apscheduler	1	2700	the framework under test is 1.08x faster than aperiodic	2.753 milliseconds
process_pool	1	2700	the framework under test is 1.09x faster than process_pool	2.753 milliseconds
apscheduler	4	40	the framework under test is 1.45x faster than aperiodic	174.59 microseconds
apscheduler	4	300	the framework under test is 1.16x faster than aperiodic	449.42 microseconds
apscheduler	16	40	the framework under test is 1.29x faster than aperiodic	177.55 microseconds

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
apscheduler	16	300	the framework under test is 1.09x faster than apscheduler	466.12 microseconds
apscheduler	64	300	the framework under test is 1.10x faster than apscheduler	462.64 microseconds

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 1.99x faster than zoo	151.05 microseconds
thread_pool	1	300	thread_pool is 1.31x faster than zoo	410.28 microseconds
bare_thread	1	2700	bare_thread is 1.05x faster than zoo	2.753 milliseconds
asyncio_pool	1	10000	asyncio_pool is 1.04x faster than zoo	9.871 milliseconds
bare_thread	1	10000	bare_thread is 1.03x faster than zoo	9.871 milliseconds
thread_pool	1	10000	thread_pool is 1.04x faster than zoo	9.871 milliseconds
thread_pool	4	40	thread_pool is 1.79x faster than zoo	174.59 microseconds
bare_thread	4	300	bare_thread is 1.09x faster than zoo	449.42 microseconds
process_pool	4	300	process_pool is 1.47x faster than zoo	449.42 microseconds
thread_pool	4	300	thread_pool is 1.30x faster than zoo	449.42 microseconds
asyncio_pool	4	2700	asyncio_pool is 1.06x faster than zoo	2.911 milliseconds
bare_thread	4	2700	bare_thread is 1.09x faster than zoo	2.911 milliseconds
process_pool	4	2700	process_pool is 2.09x faster than zoo	2.911 milliseconds
thread_pool	4	2700	thread_pool is 1.08x faster than zoo	2.911 milliseconds
asyncio_pool	4	10000	asyncio_pool is 1.03x faster than zoo	9.949 milliseconds
bare_thread	4	10000	bare_thread is 1.02x faster than zoo	9.949 milliseconds
process_pool	4	10000	process_pool is 2.14x faster than zoo	9.949 milliseconds

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
thread_pool	4	10000	thread_pool is 1.04x faster than zoo	9.949 milliseconds
asyncio_pool	16	40	asyncio_pool is 1.33x faster than zoo	177.55 microseconds
thread_pool	16	40	thread_pool is 1.97x faster than zoo	177.55 microseconds
asyncio_pool	16	300	asyncio_pool is 1.18x faster than zoo	466.12 microseconds
bare_thread	16	300	bare_thread is 1.10x faster than zoo	466.12 microseconds
process_pool	16	300	process_pool is 1.69x faster than zoo	466.12 microseconds
thread_pool	16	300	thread_pool is 1.35x faster than zoo	466.12 microseconds
asyncio_pool	16	2700	asyncio_pool is 1.05x faster than zoo	2.908 milliseconds
bare_thread	16	2700	bare_thread is 1.08x faster than zoo	2.908 milliseconds
process_pool	16	2700	process_pool is 2.11x faster than zoo	2.908 milliseconds
thread_pool	16	2700	thread_pool is 1.06x faster than zoo	2.908 milliseconds
bare_thread	16	10000	bare_thread is 1.04x faster than zoo	10.146 milliseconds
process_pool	16	10000	process_pool is 2.15x faster than zoo	10.146 milliseconds
thread_pool	16	10000	thread_pool is 1.04x faster than zoo	10.146 milliseconds
asyncio_pool	64	40	asyncio_pool is 1.68x faster than zoo	205.09 microseconds
bare_thread	64	40	bare_thread is 1.20x faster than zoo	205.09 microseconds
thread_pool	64	40	thread_pool is 2.24x faster than zoo	205.09 microseconds
asyncio_pool	64	300	asyncio_pool is 1.21x faster than zoo	462.64 microseconds
bare_thread	64	300	bare_thread is 1.09x faster than zoo	462.64 microseconds
process_pool	64	300	process_pool is 1.51x faster than zoo	462.64 microseconds
thread_pool	64	300	thread_pool is 1.31x faster than zoo	462.64 microseconds
asyncio_pool	64	2700	asyncio_pool is 1.04x faster than zoo	2.884 milliseconds
bare_thread	64	2700	bare_thread is 1.05x faster than zoo	2.884 milliseconds
process_pool	64	2700	process_pool is 2.09x faster than zoo	2.884 milliseconds

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
thread_pool	64	2700	thread_pool is 1.09x faster than zoo	2.884 milliseconds
asyncio_pool	64	10000	asyncio_pool is 1.06x faster than zoo	10.447 milliseconds
bare_thread	64	10000	bare_thread is 1.10x faster than zoo	10.447 milliseconds
process_pool	64	10000	process_pool is 2.22x faster than zoo	10.447 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: 1.3% to 50.0% (it varies with each tier's spread).

baseline	concurrency	body tier (microseconds)	ratio	reason
asyncio_pool	1	40	0.94x	the two sides differ by 5.8%, less than the band of 19.9% - no winner
bare_thread	1	40	0.96x	the two sides differ by 3.9%, less than the band of 19.5% - no winner
asyncio_pool	1	300	0.94x	the two sides differ by 6.1%, less than the band of 7.9% - no winner
bare_thread	1	300	0.95x	the two sides differ by 5.4%, less than the band of 7.7% - no winner
asyncio_pool	1	2700	0.98x	the two sides differ by 2.0%, less than the band of 3.2% - no winner
thread_pool	1	2700	0.97x	the two sides differ by 3.2%, less than the band of 3.4% - no winner

baseline	concurrency	body tier (microseconds)	ratio	reason
apscheduler	1	10000	1.01x	the two sides differ by 0.7%, less than the band of 1.3% - no winner
process_pool	1	10000	0.99x	the two sides differ by 1.0%, less than the band of 2.6% - no winner
asyncio_pool	4	40	0.93x	the two sides differ by 6.9%, less than the band of 21.6% - no winner
bare_thread	4	40	0.91x	the two sides differ by 9.4%, less than the band of 13.9% - no winner
process_pool	4	40	0.96x	the two sides differ by 4.2%, less than the band of 31.7% - no winner
asyncio_pool	4	300	0.94x	the two sides differ by 6.0%, less than the band of 7.6% - no winner
apscheduler	4	2700	1.00x	the two sides differ by 0.1%, less than the band of 2.4% - no winner
apscheduler	4	10000	1.00x	the two sides differ by 0.4%, less than the band of 1.7% - no winner
bare_thread	16	40	0.96x	the two sides differ by 3.5%, less than the band of 14.8% - no winner
process_pool	16	40	0.88x	the two sides differ by 11.9%, less than the band of 27.7% - no winner
apscheduler	16	2700	1.01x	the two sides differ by 1.3%, less than the band of 2.4% - no winner
apscheduler	16	10000	1.01x	the two sides differ by 0.7%, less than the band of 1.8% - no winner
asyncio_pool	16	10000	0.98x	the two sides differ by 2.3%, less than the band of 3.0% - no winner

baseline	concurrency	body tier (microseconds)	ratio	reason
apscheduler	64	40	1.08x	the two sides differ by 7.8%, less than the band of 14.3% - no winner
process_pool	64	40	1.00x	the two sides differ by 0.5%, less than the band of 17.2% - no winner
apscheduler	64	2700	1.01x	the two sides differ by 0.7%, less than the band of 4.1% - no winner
apscheduler	64	10000	0.98x	the two sides differ by 2.4%, less than the band of 4.9% - no winner
thread_pool	64	10000	0.98x	the two sides differ by 2.0%, less than the band of 4.0% - 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': 85458, 'child_pids': [85914, 85917, 85920, 85923, 85926, 85932, 85938, 85944, 85950, 85968, 85986, 86004, 86022, 86079, 86137, 86193, 86260, 86262, 86264, 86266, 86268, 86273, 86278, 86283, 86288, 86305, 86322, 86339, 86419, 86478, 86538, 86604, 86671, 86705, 86739, 86773, 86807, 86940, 87073, 87206, 87339, 87868, 88397, 88926, 89455, 91569, 93682, 95795, 97911, 97915, 97919, 97923, 97927, 97934, 97941, 97948, 97955, 97974, 97993, 98012, 98031, 98098, 98165, 98232, 98301, 98303, 98305, 98307, 98309, 98314, 98319, 98324, 98329, 98346, 98363, 98380, 98397, 98453, 98510, 98570, 98636, 98638, 98640, 98642, 98644, 98649, 98654, 98659, 98664, 98681, 98698, 98715, 98732, 98797, 98862, 98928], '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	287.86 microseconds	333.65 microseconds	337.30 microseconds	0.060
apscheduler	1	300	553.65 microseconds	580.37 microseconds	582.89 microseconds	0.031
apscheduler	1	2700	2.987 milliseconds	3.023 milliseconds	3.035 milliseconds	0.008
apscheduler	1	10000	9.941 milliseconds	10.079 milliseconds	10.094 milliseconds	0.007
apscheduler	4	40	252.70 microseconds	263.15 microseconds	300.38 microseconds	0.049
apscheduler	4	300	521.58 microseconds	538.80 microseconds	577.86 microseconds	0.028
apscheduler	4	2700	2.912 milliseconds	2.962 milliseconds	2.986 milliseconds	0.011
apscheduler	4	10000	9.993 milliseconds	10.119 milliseconds	10.152 milliseconds	0.008
apscheduler	16	40	229.36 microseconds	269.71 microseconds	274.62 microseconds	0.067
apscheduler	16	300	505.80 microseconds	527.94 microseconds	535.19 microseconds	0.028
apscheduler	16	2700	2.947 milliseconds	2.982 milliseconds	3.001 milliseconds	0.013
apscheduler	16	10000	10.216 milliseconds	10.299 milliseconds	10.606 milliseconds	0.011
apscheduler	64	40	221.11 microseconds	248.41 microseconds	274.24 microseconds	0.061
apscheduler	64	300	509.59 microseconds	529.92 microseconds	536.65 microseconds	0.025
apscheduler	64	2700	2.905 milliseconds	2.962 milliseconds	3.045 milliseconds	0.014
apscheduler	64	10000	10.197 milliseconds	10.650 milliseconds	10.669 milliseconds	0.025
asyncio_pool	1	40	142.30 microseconds	169.08 microseconds	183.35 microseconds	0.089
asyncio_pool	1	300	385.12 microseconds	413.86 microseconds	420.57 microseconds	0.038
asyncio_pool	1	2700	2.699 milliseconds	2.736 milliseconds	2.766 milliseconds	0.008
asyncio_pool	1	10000	9.474 milliseconds	9.637 milliseconds	9.654 milliseconds	0.007
asyncio_pool	4	40	162.47 microseconds	183.07 microseconds	190.87 microseconds	0.107
asyncio_pool	4	300	422.43 microseconds	446.86 microseconds	454.55 microseconds	0.035
asyncio_pool	4	2700	2.754 milliseconds	2.798 milliseconds	2.816 milliseconds	0.007
asyncio_pool	4	10000	9.670 milliseconds	9.780 milliseconds	9.784 milliseconds	0.007

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
asyncio_pool	16	40	133.81 microseconds	149.95 microseconds	153.63 microseconds	0.057
asyncio_pool	16	300	396.45 microseconds	440.03 microseconds	451.15 microseconds	0.041
asyncio_pool	16	2700	2.770 milliseconds	2.814 milliseconds	2.838 milliseconds	0.010
asyncio_pool	16	10000	9.917 milliseconds	10.414 milliseconds	10.835 milliseconds	0.024
asyncio_pool	64	40	121.88 microseconds	125.59 microseconds	126.51 microseconds	0.022
asyncio_pool	64	300	381.72 microseconds	389.63 microseconds	391.25 microseconds	0.012
asyncio_pool	64	2700	2.779 milliseconds	2.814 milliseconds	2.838 milliseconds	0.008
asyncio_pool	64	10000	9.811 milliseconds	10.103 milliseconds	10.291 milliseconds	0.019
bare_thread	1	40	145.22 microseconds	170.23 microseconds	187.14 microseconds	0.085
bare_thread	1	300	388.13 microseconds	414.02 microseconds	417.51 microseconds	0.036
bare_thread	1	2700	2.634 milliseconds	2.671 milliseconds	2.689 milliseconds	0.008
bare_thread	1	10000	9.553 milliseconds	9.678 milliseconds	9.949 milliseconds	0.011
bare_thread	4	40	158.14 microseconds	166.34 microseconds	167.15 microseconds	0.030
bare_thread	4	300	412.54 microseconds	430.76 microseconds	437.51 microseconds	0.021
bare_thread	4	2700	2.683 milliseconds	2.720 milliseconds	2.791 milliseconds	0.011
bare_thread	4	10000	9.713 milliseconds	9.847 milliseconds	10.230 milliseconds	0.014
bare_thread	16	40	171.28 microseconds	177.34 microseconds	182.74 microseconds	0.021
bare_thread	16	300	423.99 microseconds	433.21 microseconds	435.95 microseconds	0.011
bare_thread	16	2700	2.705 milliseconds	2.720 milliseconds	2.725 milliseconds	0.004
bare_thread	16	10000	9.716 milliseconds	9.841 milliseconds	10.636 milliseconds	0.023
bare_thread	64	40	170.85 microseconds	175.30 microseconds	177.98 microseconds	0.017
bare_thread	64	300	425.01 microseconds	440.28 microseconds	446.19 microseconds	0.017
bare_thread	64	2700	2.746 milliseconds	2.773 milliseconds	2.785 milliseconds	0.005
bare_thread	64	10000	9.532 milliseconds	9.573 milliseconds	9.611 milliseconds	0.003

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
process_pool	1	40	349.41 microseconds	373.59 microseconds	885.09 microseconds	0.390
process_pool	1	300	590.88 microseconds	645.66 microseconds	1.134 milliseconds	0.232
process_pool	1	2700	2.988 milliseconds	3.035 milliseconds	3.660 milliseconds	0.057
process_pool	1	10000	9.773 milliseconds	9.841 milliseconds	10.532 milliseconds	0.020
process_pool	4	40	167.32 microseconds	253.53 microseconds	269.90 microseconds	0.208
process_pool	4	300	306.62 microseconds	387.79 microseconds	389.90 microseconds	0.130
process_pool	4	2700	1.390 milliseconds	1.534 milliseconds	1.906 milliseconds	0.092
process_pool	4	10000	4.653 milliseconds	5.019 milliseconds	5.539 milliseconds	0.047
process_pool	16	40	156.46 microseconds	196.85 microseconds	224.41 microseconds	0.150
process_pool	16	300	275.04 microseconds	316.76 microseconds	328.71 microseconds	0.083
process_pool	16	2700	1.381 milliseconds	1.503 milliseconds	1.512 milliseconds	0.040
process_pool	16	10000	4.712 milliseconds	4.885 milliseconds	5.054 milliseconds	0.020
process_pool	64	40	204.14 microseconds	231.60 microseconds	239.49 microseconds	0.090
process_pool	64	300	307.25 microseconds	337.78 microseconds	364.57 microseconds	0.057
process_pool	64	2700	1.379 milliseconds	1.423 milliseconds	1.437 milliseconds	0.014
process_pool	64	10000	4.713 milliseconds	4.884 milliseconds	5.023 milliseconds	0.018
thread_pool	1	40	75.73 microseconds	93.57 microseconds	98.40 microseconds	0.090
thread_pool	1	300	313.96 microseconds	346.42 microseconds	348.61 microseconds	0.041
thread_pool	1	2700	2.667 milliseconds	2.707 milliseconds	2.742 milliseconds	0.009
thread_pool	1	10000	9.525 milliseconds	9.775 milliseconds	9.801 milliseconds	0.012
thread_pool	4	40	97.54 microseconds	106.95 microseconds	113.51 microseconds	0.057
thread_pool	4	300	346.42 microseconds	359.06 microseconds	360.93 microseconds	0.017
thread_pool	4	2700	2.689 milliseconds	2.742 milliseconds	2.748 milliseconds	0.011
thread_pool	4	10000	9.529 milliseconds	9.682 milliseconds	9.693 milliseconds	0.009

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
thread_pool	16	40	90.20 microseconds	102.75 microseconds	115.38 microseconds	0.077
thread_pool	16	300	345.34 microseconds	359.59 microseconds	363.79 microseconds	0.019
thread_pool	16	2700	2.740 milliseconds	2.791 milliseconds	2.801 milliseconds	0.010
thread_pool	16	10000	9.785 milliseconds	9.874 milliseconds	9.995 milliseconds	0.007
thread_pool	64	40	91.39 microseconds	93.01 microseconds	93.77 microseconds	0.014
thread_pool	64	300	352.74 microseconds	356.45 microseconds	358.38 microseconds	0.007
thread_pool	64	2700	2.647 milliseconds	2.680 milliseconds	2.722 milliseconds	0.011
thread_pool	64	10000	10.238 milliseconds	10.321 milliseconds	10.366 milliseconds	0.016
zoo	1	40	151.05 microseconds	196.92 microseconds	200.72 microseconds	0.110
zoo	1	300	410.28 microseconds	442.59 microseconds	449.04 microseconds	0.041
zoo	1	2700	2.753 milliseconds	2.812 milliseconds	3.016 milliseconds	0.024
zoo	1	10000	9.871 milliseconds	9.967 milliseconds	9.991 milliseconds	0.006
zoo	4	40	174.59 microseconds	222.70 microseconds	232.01 microseconds	0.109
zoo	4	300	449.42 microseconds	484.37 microseconds	490.04 microseconds	0.041
zoo	4	2700	2.911 milliseconds	2.977 milliseconds	2.995 milliseconds	0.013
zoo	4	10000	9.949 milliseconds	10.060 milliseconds	10.074 milliseconds	0.008
zoo	16	40	177.55 microseconds	227.24 microseconds	234.50 microseconds	0.126
zoo	16	300	466.12 microseconds	502.80 microseconds	521.18 microseconds	0.047
zoo	16	2700	2.908 milliseconds	2.966 milliseconds	2.977 milliseconds	0.011
zoo	16	10000	10.146 milliseconds	10.244 milliseconds	10.263 milliseconds	0.007
zoo	64	40	205.09 microseconds	217.27 microseconds	218.65 microseconds	0.082
zoo	64	300	462.64 microseconds	479.84 microseconds	481.79 microseconds	0.020
zoo	64	2700	2.884 milliseconds	3.017 milliseconds	3.174 milliseconds	0.027
zoo	64	10000	10.447 milliseconds	10.600 milliseconds	10.612 milliseconds	0.024

Appendix: Framework overhead share

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
apscheduler	1	40	237.28 microseconds	82.43%	50.59 microseconds
apscheduler	1	300	254.40 microseconds	45.95%	299.25 microseconds
apscheduler	1	2700	284.31 microseconds	9.52%	2.703 milliseconds
apscheduler	1	10000	347.66 microseconds	3.50%	9.594 milliseconds
apscheduler	4	40	197.07 microseconds	77.98%	55.63 microseconds
apscheduler	4	300	208.67 microseconds	40.01%	312.91 microseconds
apscheduler	4	2700	269.47 microseconds	9.25%	2.643 milliseconds
apscheduler	4	10000	-	-	14.685 milliseconds
apscheduler	16	40	-	-	53.56 microseconds
apscheduler	16	300	-	-	312.07 microseconds
apscheduler	16	2700	-	-	2.691 milliseconds
apscheduler	16	10000	-	-	20.189 milliseconds
apscheduler	64	40	-	-	50.03 microseconds
apscheduler	64	300	-	-	303.93 microseconds
apscheduler	64	2700	-	-	2.650 milliseconds
apscheduler	64	10000	-	-	19.926 milliseconds
asyncio_pool	1	40	95.76 microseconds	67.30%	46.53 microseconds
asyncio_pool	1	300	106.93 microseconds	27.77%	278.19 microseconds
asyncio_pool	1	2700	118.47 microseconds	4.39%	2.581 milliseconds
asyncio_pool	1	10000	161.66 microseconds	1.71%	9.313 milliseconds
asyncio_pool	4	40	109.84 microseconds	67.61%	52.62 microseconds
asyncio_pool	4	300	113.36 microseconds	26.84%	309.07 microseconds
asyncio_pool	4	2700	151.00 microseconds	5.48%	2.603 milliseconds
asyncio_pool	4	10000	-	-	13.998 milliseconds
asyncio_pool	16	40	-	-	53.67 microseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
asyncio_pool	16	300	-	-	303.60 microseconds
asyncio_pool	16	2700	-	-	2.649 milliseconds
asyncio_pool	16	10000	-	-	23.932 milliseconds
asyncio_pool	64	40	-	-	51.56 microseconds
asyncio_pool	64	300	-	-	301.54 microseconds
asyncio_pool	64	2700	-	-	2.653 milliseconds
asyncio_pool	64	10000	-	-	25.039 milliseconds
bare_thread	1	40	93.84 microseconds	64.62%	51.39 microseconds
bare_thread	1	300	102.79 microseconds	26.48%	285.34 microseconds
bare_thread	1	2700	111.50 microseconds	4.23%	2.523 milliseconds
bare_thread	1	10000	129.62 microseconds	1.36%	9.423 milliseconds
bare_thread	4	40	108.38 microseconds	68.53%	49.76 microseconds
bare_thread	4	300	115.37 microseconds	27.97%	297.17 microseconds
bare_thread	4	2700	131.26 microseconds	4.89%	2.551 milliseconds
bare_thread	4	10000	-	-	14.689 milliseconds
bare_thread	16	40	-	-	51.26 microseconds
bare_thread	16	300	-	-	293.98 microseconds
bare_thread	16	2700	-	-	2.573 milliseconds
bare_thread	16	10000	-	-	14.962 milliseconds
bare_thread	64	40	-	-	51.28 microseconds
bare_thread	64	300	-	-	296.98 microseconds
bare_thread	64	2700	-	-	2.611 milliseconds
bare_thread	64	10000	-	-	14.624 milliseconds
process_pool	1	40	294.27 microseconds	84.22%	55.13 microseconds
process_pool	1	300	291.48 microseconds	49.33%	299.40 microseconds
process_pool	1	2700	338.29 microseconds	11.32%	2.650 milliseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
process_pool	1	10000	356.64 microseconds	3.65%	9.416 milliseconds
process_pool	4	40	72.43 microseconds	43.29%	94.89 microseconds
process_pool	4	300	-	-	565.29 microseconds
process_pool	4	2700	-	-	4.828 milliseconds
process_pool	4	10000	-	-	17.843 milliseconds
process_pool	16	40	-	-	91.42 microseconds
process_pool	16	300	-	-	569.51 microseconds
process_pool	16	2700	-	-	9.767 milliseconds
process_pool	16	10000	-	-	50.238 milliseconds
process_pool	64	40	-	-	93.94 microseconds
process_pool	64	300	-	-	580.85 microseconds
process_pool	64	2700	-	-	27.218 milliseconds
process_pool	64	10000	-	-	167.379 milliseconds
thread_pool	1	40	31.18 microseconds	41.18%	44.55 microseconds
thread_pool	1	300	35.34 microseconds	11.25%	278.63 microseconds
thread_pool	1	2700	48.08 microseconds	1.80%	2.618 milliseconds
thread_pool	1	10000	57.18 microseconds	0.60%	9.467 milliseconds
thread_pool	4	40	47.22 microseconds	48.41%	50.32 microseconds
thread_pool	4	300	52.48 microseconds	15.15%	293.94 microseconds
thread_pool	4	2700	94.29 microseconds	3.51%	2.595 milliseconds
thread_pool	4	10000	-	-	16.292 milliseconds
thread_pool	16	40	-	-	47.40 microseconds
thread_pool	16	300	-	-	295.01 microseconds
thread_pool	16	2700	-	-	2.665 milliseconds
thread_pool	16	10000	-	-	20.027 milliseconds
thread_pool	64	40	-	-	47.44 microseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
thread_pool	64	300	-	-	292.76 microseconds
thread_pool	64	2700	-	-	2.552 milliseconds
thread_pool	64	10000	-	-	45.814 milliseconds
zoo	1	40	102.60 microseconds	67.93%	48.45 microseconds
zoo	1	300	115.00 microseconds	28.03%	295.27 microseconds
zoo	1	2700	137.57 microseconds	5.00%	2.616 milliseconds
zoo	1	10000	160.70 microseconds	1.63%	9.711 milliseconds
zoo	4	40	126.39 microseconds	72.39%	48.20 microseconds
zoo	4	300	149.04 microseconds	33.16%	300.39 microseconds
zoo	4	2700	176.55 microseconds	6.07%	2.734 milliseconds
zoo	4	10000	-	-	15.052 milliseconds
zoo	16	40	-	-	44.76 microseconds
zoo	16	300	-	-	315.32 microseconds
zoo	16	2700	-	-	2.751 milliseconds
zoo	16	10000	-	-	20.587 milliseconds
zoo	64	40	-	-	49.05 microseconds
zoo	64	300	-	-	293.02 microseconds
zoo	64	2700	-	-	2.719 milliseconds
zoo	64	10000	-	-	102.369 milliseconds

Appendix: Throughput and concurrency scaling

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
apscheduler	1	40	3473.9
apscheduler	1	300	1806.2
apscheduler	1	2700	334.8
apscheduler	1	10000	100.6
apscheduler	4	40	15828.8
apscheduler	4	300	7669.0
apscheduler	4	2700	1373.4
apscheduler	4	10000	400.3
apscheduler	16	40	69759.2

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
apscheduler	16	300	31632.8
apscheduler	16	2700	5429.2
apscheduler	16	10000	1566.2
apscheduler	64	40	289446.9
apscheduler	64	300	125590.4
apscheduler	64	2700	22031.0
apscheduler	64	10000	6276.1
asyncio_pool	1	40	7027.6
asyncio_pool	1	300	2596.6
asyncio_pool	1	2700	370.4
asyncio_pool	1	10000	105.5
asyncio_pool	4	40	24620.6
asyncio_pool	4	300	9468.9
asyncio_pool	4	2700	1452.5
asyncio_pool	4	10000	413.6
asyncio_pool	16	40	119571.2
asyncio_pool	16	300	40357.7
asyncio_pool	16	2700	5775.3
asyncio_pool	16	10000	1613.4
asyncio_pool	64	40	525124.2
asyncio_pool	64	300	167662.6
asyncio_pool	64	2700	23032.2
asyncio_pool	64	10000	6523.5
bare_thread	1	40	6886.0
bare_thread	1	300	2576.4
bare_thread	1	2700	379.6
bare_thread	1	10000	104.7
bare_thread	4	40	25293.9
bare_thread	4	300	9696.1
bare_thread	4	2700	1491.0
bare_thread	4	10000	411.8
bare_thread	16	40	93414.7
bare_thread	16	300	37737.0
bare_thread	16	2700	5915.8
bare_thread	16	10000	1646.8
bare_thread	64	40	374606.5
bare_thread	64	300	150584.2
bare_thread	64	2700	23310.3
bare_thread	64	10000	6714.2
process_pool	1	40	2862.0
process_pool	1	300	1692.4
process_pool	1	2700	334.6
process_pool	1	10000	102.3
process_pool	4	40	23906.0

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
process_pool	4	300	13045.3
process_pool	4	2700	2878.5
process_pool	4	10000	859.6
process_pool	16	40	102262.4
process_pool	16	300	58172.4
process_pool	16	2700	11588.2
process_pool	16	10000	3395.6
process_pool	64	40	313503.0
process_pool	64	300	208301.1
process_pool	64	2700	46410.5
process_pool	64	10000	13578.0
thread_pool	1	40	13204.5
thread_pool	1	300	3185.1
thread_pool	1	2700	375.0
thread_pool	1	10000	105.0
thread_pool	4	40	41009.0
thread_pool	4	300	11546.7
thread_pool	4	2700	1487.4
thread_pool	4	10000	419.8
thread_pool	16	40	177375.4
thread_pool	16	300	46331.7
thread_pool	16	2700	5838.4
thread_pool	16	10000	1635.2
thread_pool	64	40	700311.8
thread_pool	64	300	181435.9
thread_pool	64	2700	24173.8
thread_pool	64	10000	6251.0
zoo	1	40	6620.4
zoo	1	300	2437.4
zoo	1	2700	363.2
zoo	1	10000	101.3
zoo	4	40	22910.9
zoo	4	300	8900.3
zoo	4	2700	1374.2
zoo	4	10000	402.1
zoo	16	40	90116.9
zoo	16	300	34326.2
zoo	16	2700	5501.6
zoo	16	10000	1577.0
zoo	64	40	312054.2
zoo	64	300	138336.3
zoo	64	2700	22191.3
zoo	64	10000	6126.4