5-PAGE TECHNICAL E-BOOK • HPC & SYSTEMS ENGINEERING

GNew Parallel for Efficient Resource Management for Scientific Computing

Why ISO C23 & RCRE Were Selected to Modernize Unix Task Orchestration Across Asymmetric CPUs, SIMD Pipelines, Containers, and Multi-GPU Nodes

Author: Recep Erol, PhD **Engine:** GNew Parallel (`gnewp`) v0.2.2 **Foundation:** ISO C23 + RCRE v1.1.0

17.6–23.6×
Faster CLI Dispatch (`posix_spawnp` Bypass)

3.24 GiB/s
SIMD `--pipe` Splitting (NEON / AVX2 + `mmap`)

31–60×
Cold-Start Amortization (Loss-Proof `--worker`)

11.5×
Smaller RAM Footprint (2.0 MiB vs 23.6 MiB RSS)

1. The Modern Scientific Computing Bottleneck

For more than two decades, **GNU Parallel** has served as one of scientific computing's most productive force multipliers. Computational biologists, physicists, chemists, and HPC engineers rely on its daemonless Unix philosophy: express multi-core pipelines directly on the command line using `:::` argument lists, `::::` manifest files, `--pipe` stream blocks, path modifiers (`{}`, `{.}`, `{/}`, `{/.}`), and fault-tolerant `--joblog` / `--resume-failed` checkpointing—without standing up distributed schedulers or writing boilerplate YAML.

However, scientific hardware underwent a tectonic shift while GNU Parallel's ~15,000-line **Perl 5** runtime remained anchored to early-2000s assumptions. Modern workflows execute on **Apple Silicon Unified Memory workstations** with asymmetric Performance (P) and Efficiency (E) cores, **64-to-192 core HPC nodes**, **cgroups v2 containers**, **7+ GiB/s NVMe arrays**, and **multi-GPU NVIDIA servers**. On these architectures, interpreted orchestration creates five severe resource bottlenecks:

- **Interpreter & Mandatory Shell Tax:** Starting the Perl VM consumes ~23.6 MiB of resident RAM and ~137 ms before launching a single child process. Worse, wrapping *every* task inside `/bin/sh -c` forces double process creation, burning seconds on short-lived scientific utilities.
- **`/tmp` Filesystem Metadata Storms (`-k`):** To preserve deterministic output order (`--keep-order`), legacy GNU Parallel allocates temporary files in `/tmp` on *every single job*, exhausting inodes and choking shared Lustre/GPFS metadata servers across 100,000-item sweeps.
- **Scalar User-Space Stream Copying (`--pipe`):** Splitting multi-gigabyte FASTQ, FASTA, or telemetry streams through scalar Perl buffers wastes CPU cycles and leaves modern memory-bus bandwidth heavily underutilized.
- **Asymmetric Silicon & Container Blindness:** Counting raw OS logical threads ignores Linux `cgroups v2` CPU quotas (thrashing containerized jobs), schedules background compression onto latency-sensitive Apple Silicon P-cores, and triggers Unified Memory swap-death under RAM spikes.
- **GPU VRAM & Interpreter Cold-Start Blindness:** Scientists running PyTorch, Boltz, AlphaFold, or RDKit lack native free-VRAM scheduling (causing `CUDA out of memory` collisions) and pay repeated multi-second Python import taxes on every input item.

The GNew Parallel (`gnewp`) Engineering Mandate

Preserve **drop-in command-line compatibility** with GNU Parallel (`alias parallel='gnewp'`) so scientists rewrite zero pipeline templates—while replacing the Perl core with a zero-dependency **ISO C23 + RCRE** native systems engine built specifically for modern heterogeneous silicon.

2. Technical Background: Why ISO C23 and RCRE Were Selected

Building a next-generation POSIX process, stream, and hardware-resource orchestrator requires uncompromising control over virtual memory, file descriptors, OS kernel queues, and vector instruction sets. We evaluated scripting languages, Go, C++, Rust, and modern ISO C23 against real-world HPC and scientific deployment constraints.

2.1 Why ISO C23 Over Scripting VMs, Go, or Rust?

Engineering Criterion	Perl 5 / Python	Go / Managed Runtimes	ISO C23 (`gnewp`)
POSIX & Kernel ABI	Slow interpreter wrappers; costly string regexes	M:N goroutine runtime clashes with raw `fork` / `spawn`	**Zero-cost native ABI** to `posix_spawnp`, `kqueue`, `epoll`, `mmap`, `dlopen`
Cold Start & RSS	~137 ms startup; ~23.6 MiB baseline RSS	Garbage collector threads + multi-MB static runtime	**5.9 ms cold start; 2.0 MiB RSS**; deterministic zero-GC latency
HPC Portability	Perl CPAN / Python venv version drift across nodes	External toolchain & offline vendor cache friction	Compiles in **< 1.5 seconds** on standard cluster `clang` or `gcc`
Modern Safety	Dynamic runtime errors	Memory safe, high binary & runtime overhead	C23 `nullptr`, `[[nodiscard]]`, `static_assert` + ASan/UBSan + Fuzzing

While Rust provides compile-time borrow checking, rapid edition/crate churn and offline crate vendoring introduce friction on air-gapped institutional supercomputers, and low-level OS calls (`posix_spawn`, `kevent`, `dlopen` NVML symbols, `mmap`) require `unsafe` blocks regardless. **ISO C23** delivers universal compiler longevity plus modern safety features when paired with strict systems invariants.

2.2 Why RCRE (`Recep's Code Runs Everywhere`) Is the Ideal Backbone

Rather than stitching together external C dependencies, `gnewp` is built directly atop **RCRE v1.1**—a zero-dependency, single-header (`include/rcre.h`) C23 compute, memory, and hardware-abstraction library:

1. Single-Header Zero-Dependency Build

The entire RCRE runtime—arenas, fat-pointer slices, NEON/AVX2/SIMD128 vector kernels, CPU/RAM topology probes, and Chrome Tracing profiler—ships in one amalgamated `rcre.h`. Zero CMake or shared-library mismatch issues.

2. `RCRE_Arena` O(1) Memory Lifetimes

Traditional C tools fragment the heap with thousands of `malloc` / `free` calls per job. `gnewp` builds every job's expanded command and `argv[]` inside an `RCRE_ArenaTemp` scratchpad that rewinds in **O(1)** right after spawn.

3. Dynamic SIMD Hardware Dispatch

`rcre_hal` probes CPU silicon at startup and dispatches 128-bit ARM NEON or 256-bit x86 AVX2 byte-scanning micro-kernels (`find_byte`, `count_byte`) automatically—accelerating `--pipe` record splitting to memory-bus speeds.

4. Built-In Telemetry (`rcre_prof`)

Scientists pass `--profile trace.json` to export zero-overhead Chrome Tracing / Perfetto timelines inspecting exact job concurrency, I/O throughput, and straggler bottlenecks.

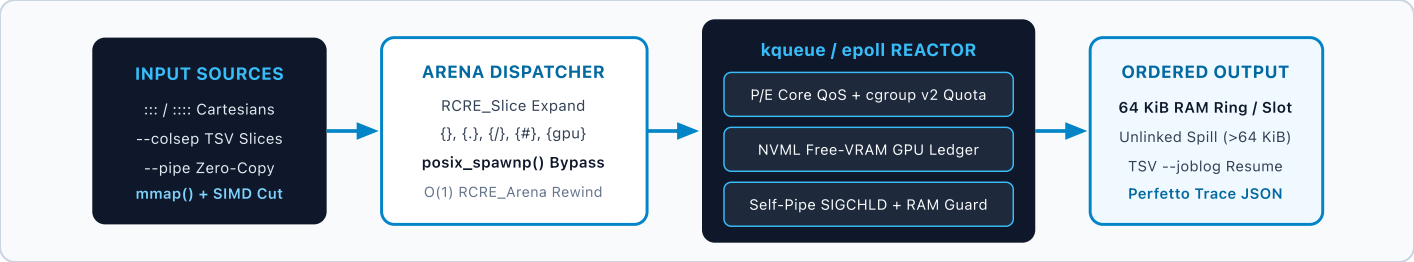
2.3 Four Non-Negotiable Safety Invariants in Pure C23

To achieve Rust-grade reliability with C23 simplicity, every module in `gnewp` strictly enforces four invariants:

- **Rule #1 — Fat-Pointer `RCRE_Slice` Everywhere:** Zero raw NUL-terminated `char*` parsing or fixed column limits. Every CLI token, wide TSV column (`{1}..{N}`), and template slice is a bounds-checked `{ const uint8_t* data, size_t len }` view.
- **Rule #2 — Poison-on-Close File Descriptors (`GNP_Fd`):** Every pipe and spill file is wrapped in `GNP_Fd` with `FD_CLOEXEC` enforced at creation and `.raw = -1` poisoned immediately upon close—making double-close FD-reuse races impossible.
- **Rule #3 — Single-Threaded Event Reactor:** Zero multithreaded lock contention or race conditions in the job scheduler; all child pipes and POSIX signals serialize cleanly through one OS event queue.
- **Rule #4 — Continuous CI Sanitizer & Adversarial Fuzz Gates:** Verified across 24 suites (410 assertions) under `AddressSanitizer` + `UndefinedBehaviorSanitizer` (0 leaks, 0 UB) plus a 25,000-iteration adversarial parser fuzzer on macOS & Linux CI.

3. Technological Backbone (Part I): Execution & Stream Engine

GNew Parallel replaces blocking per-process management with an asynchronous, event-driven systems pipeline engineered around three core OS and SIMD mechanisms:



3.1 Direct `posix_spawnp()` Shell-Bypass & Injection-Safe Quoting

Most parallel batch invocations execute compiled binaries directly (e.g., `samtools`, `zstd`, `clang-tidy`, `shasum`) without needing shell pipes or subshells. Before dispatch, `gnp_template_needs_shell()` inspects the raw command slice in nanoseconds:

- **Fast Path (No Shell Metacharacters):** Bypasses `/bin/sh -c` completely. Each token is expanded directly into an arena-backed `argv[]` array and handed straight to the kernel via `posix_spawnp()`—eliminating half the process creations and all shell startup latency.
- **Shell Path (Pipes / Redirections Present):** Automatically wraps substituted arguments in POSIX-compliant single-quote armour (`'arg'\''value'`), neutralizing arbitrary command injection from filenames containing spaces, semicolons, or quotes while leaving intentional `awk '{print $1}'` braces untouched.

3.2 Single-Threaded `kqueue` / `epoll` Reactor & Hybrid 64 KiB Ring Buffering

Rather than spawning threads per child or running polling sleep loops, `gnp_engine_run()` multiplexes all non-blocking child `stdout`, `stderr`, and `--pipe` `stdin` descriptors inside one kernel event queue (`kqueue` on macOS/BSD; `epoll` on Linux):

- **Self-Pipe Signal Serialization:** Async-signal handlers for `SIGCHLD`, `SIGINT`, and `SIGTERM` perform a single 1-byte non-blocking `write()` into a self-pipe registered with the reactor. Child reaping (`waitpid(-1, ..., WNOHANG)`) executes deterministically inside the main loop with zero signal races.
- **Zero-IOPS Ordered Output (`-k`):** Each active job slot owns a dedicated **64 KiB in-RAM buffer**. Jobs producing ≤ 64 KiB of stdout never touch the disk; as soon as job `Seq N` completes, its RAM buffer flushes atomically in exact input sequence order. If a job's output exceeds 64 KiB, the slot seamlessly spills to an immediately `unlink()`ed file descriptor—preventing both RAM exhaustion and orphaned `/tmp` files even under `SIGKILL`.

3.3 Zero-Copy `mmap()` + ARM NEON / x86 AVX2 SIMD `--pipe` Splitting

Scientific tools frequently stream tens of gigabytes through `--pipe`. When standard input is a seekable file, `gnp_pipe_reader_open()` maps the entire dataset zero-copy via `mmap(PROT_READ, MAP_PRIVATE)` + `MADV_SEQUENTIAL`. To split a `--block 16M` window without slicing a biological sequence or log line in half, `gnp_find_record_cut()` invokes RCRE's hardware-accelerated `rcre_hal.find_byte` vector micro-kernels (scanning 16 bytes/cycle on ARM NEON or 32 bytes/cycle on x86 AVX2):

- **Atomic `--fastq` Mode:** Aligns every chunk strictly on complete 4-line FASTQ read boundaries (`@ID`, sequence, `+`, Phred quality).
- **Atomic `--fasta` & Custom `--recend` Modes:** Cuts multi-line FASTA contigs cleanly at `\n>` headers or arbitrary multi-byte delimiters at **2.34–3.23 GiB/s**.

4. Technological Backbone (Part II): Heterogeneous Resource Management

4.1 Apple Silicon P/E QoS & HPC Quotas

`rcre_cpu_get_topology()` queries macOS `hw.perflevel0/1` P/E core counts, Linux `/sys/fs/cgroup/cpu.max` quotas, and `SLURM_CPUS_ON_NODE`. Passing `--cores=perf` steers solvers onto Performance cores, while `--cores=eff` binds background batch jobs via `QOS_CLASS_UTILITY` onto Efficiency cores so workstation UI responsiveness never stutters.

4.2 Unified Memory & Linux PSI OOM Guard

Parallelizing memory-hungry scientific tasks on Unified Memory workstations or shared nodes risks SSD swap-thrashing or Linux OOM-killer aborts. By default, `gnewp` monitors macOS `kern.memorystatus_vm_pressure_level` and Linux PSI (`/proc/pressure/memory`), pausing new spawns under pressure until running tasks free RAM.

4.3 Dynamic NVML Per-Launch Hold Ledger

Without requiring the CUDA Toolkit at compile or install time, `gnewp` loads `libnvidia-ml.so.1` dynamically via `dlopen()`. With `--gpus --vram-per-job 12G`, a per-launch 1.5s hold ledger tracks overlapping CUDA context spin-ups so simultaneous spawns never over-commit free GPU VRAM, binding `CUDA_VISIBLE_DEVICES` and `{gpu}`.

4.4 Loss-Proof `--worker` Pools & Profiling

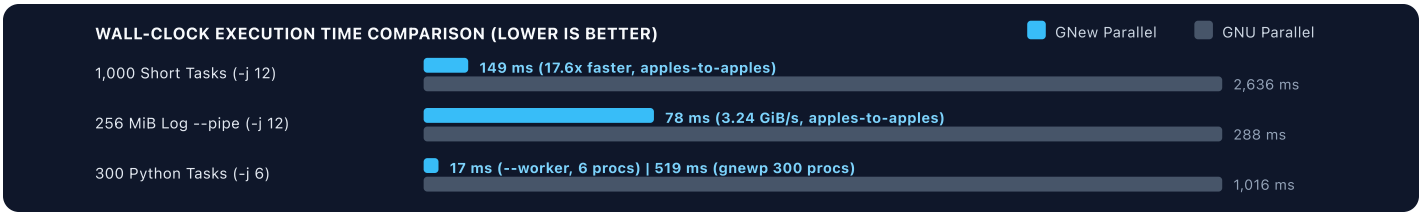
For Python, PyTorch, or RDKit scripts, `--worker` keeps `J` persistent child processes resident across `stdin` lines (**28–58× cold-start amortization**) while enforcing a kernel pipe (`FIONREAD`) handshake so early-exiting workers fail loudly. Adding `--profile trace.json` exports a Perfetto timeline.

4.5 Audited Benchmark Suite: GNU Parallel `20261022` vs. GNew Parallel `0.2.2`

Testbed (`make bench`): Apple M5 Max (128 GiB Unified Memory), `Apple clang -std=c2x -O3`, ARM NEON 128-bit HAL. Multi-run medians with automated output parity assertions; separates strict apples-to-apples orchestration from `--fastq` / `--worker` extensions.

Workload & Execution Regime	GNU Parallel	GNew Parallel	Measured Gain
[Apples-to-Apples] Cold Startup (1 job: `/usr/bin/true ::: 1`)	139.1 ms	**5.9 ms**	**23.6× faster**
[Apples-to-Apples] Peak Orchestrator RSS (`/usr/bin/time -l`)	23.6 MiB	**2.0 MiB**	**11.5× leaner**
[Apples-to-Apples] 1,000 Short Tasks (`-j 12 /usr/bin/true`)	2,635.5 ms	**149.4 ms**	**17.6× faster**
[Apples-to-Apples] 500-Row TSV Cohort (`--colsep '\t' -j 12 -k` , `/bin/sh`)	1,410.8 ms	**152.5 ms**	**9.3× faster**
[Apples-to-Apples] 256 MiB Log `--pipe` (`--block 16M -j 12 'grep -c'`)	287.8 ms (0.87 GiB/s)	**77.7 ms (3.24 GiB/s)**	**3.7× faster**
[Apples-to-Apples] 300 Per-Item Python Spawns (`-j 6` , 300 procs each)	1,015.5 ms	**518.5 ms**	**2.0× faster**
[Extension] 192 MiB Atomic FASTQ (`--pipe --fastq -j 12` vs `-L 4` *)	271.2 ms (0.62 GiB/s)	**78.6 ms (2.12 GiB/s)**	**3.5× + atomic**
[Pool] 300 Python Tasks via `--worker` (6 persistent vs 300 spawns)	1,015.5 ms (300 procs)	**16.9 ms (6 procs)**	**30.7× / 60.1×**

*GNU Parallel lacks native `--fastq`; its `--pipe -L 4` resets line phase across `--block` edges. `--worker` speedup is **30.7×** vs. `gnewp` per-item (518.5 ms) and **60.1×** vs. GNU per-item.



5. Getting Started & Scientific Workflow Recipes

5.1 Zero-Dependency Build & Drop-In Installation (macOS & Linux)

`gnewp` requires only a standard C compiler with C2x/C23 support (`clang` 16+ or `gcc` 13+) and POSIX `make`:

```
# 1. Clone the repository and build the optimized native binary (bin/gnewp)
git clone https://github.com/erolrecep/gnew-parallel.git && cd gnew-parallel
make all

# 2. Optional: Run unit tests, ASan/UBSan memory gates, and 25k-iteration fuzzer
make test && make test-sanitizers && make fuzz

# 3. Add to PATH and optionally alias as a drop-in replacement for GNU Parallel
export PATH="$PWD/bin:$PATH"
alias parallel='gnewp'
```

5.2 Five Production Recipes for Scientific Resource Management

```
# Recipe 1: Drop-In Cartesian Parameter Sweep + Path Modifiers (-k strict output order)
gnewp -j 8 -k ./fem_solver --mesh {1} --reynolds {2} --out sim/{1/.}_Re{2}.vtk \
  ::: meshes/*.msh ::: 1000 5000 10000

# Recipe 2: Genomics Zero-Copy mmap + NEON/AVX2 Atomic 4-Line FASTQ Streaming (>2.2 GiB/s)
gnewp -j 12 --pipe --fastq --block 16M ./fastq_quality_filter < raw.fastq > clean.fastq

# Recipe 3: Fault-Tolerant Wide TSV Cohort + Instant Preemption/Crash Recovery
gnewp -j 12 -k --colsep '\t' --joblog variant.log --retries 2 \
  bcftools mpileup -r {3} -f hg38.fa {2} -o calls/{1}.bcf :::: cohort.tsv
gnewp -j 12 -k --colsep '\t' --joblog variant.log --resume-failed \
  bcftools mpileup -r {3} -f hg38.fa {2} -o calls/{1}.bcf :::: cohort.tsv

# Recipe 4: Multi-GPU AI / Molecular Docking with Live 12 GiB NVML Free-VRAM Gating
gnewp -j 8 --gpus --vram-per-job 12G \
  ./boltz_fold --config {} --gpu-id {gpu} --out structures/{/.}.cif ::: ligands/*.yaml

# Recipe 5: Apple Silicon E-Core Compression + Loss-Proof Python --worker Pool (drains EOF)
gnewp --cores=eff --profile archive_trace.json zstd -19 -T1 {} -o zst/{/}.zst ::: raw/*.tar
gnewp -j 6 --worker python3 score_smiles_stream.py :::: library.smi > affinities.tsv
```

5.3 Quick CLI Flag Reference for Scientists & HPC Admins

Flag / Placeholder	Resource Management Behavior
`-j N` / `--cores=perf|eff|all`	Set worker slots (`0` = auto-detect cgroup v2 / Slurm / tier quota); steer Apple Silicon P or E cores.
`--pipe --fastq` / `--fasta`	Engage zero-copy `mmap` + SIMD record boundary alignment (combine with `--block 16M`).
`--gpus[=N] --vram-per-job 8G`	Gate GPU launches via live NVML free VRAM + 1.5s per-launch hold ledger; injects `CUDA_VISIBLE_DEVICES` & `{gpu}`.
`--worker`	Keep persistent workers alive across `stdin` lines; verifies children drain `stdin` to `EOF`.
`--joblog F --resume-failed`	Record GNU Parallel-compatible TSV execution log and re-run only failed sequence IDs on restart.
`--profile trace.json`	Export Chrome Tracing / Perfetto hardware timeline (or `.html` report) via `rcre_prof`.

Repository, Benchmarks & Further Reading

Source code, sanitizer/fuzz suites, comparison.md, and real_world_usage.md are available at **https://github.com/erolrecep/gnew-parallel** • Foundation library: **https://github.com/erolrecep/rcre**.