

## APPLICATION NOTE

# AmpliconHunter2: a SIMD-Accelerated *In-Silico* PCR Engine

Rye Howard-Stone<sup>1,\*</sup> and Ion I. Măndoiu<sup>1</sup>

<sup>1</sup>School of Computing, University of Connecticut, 371 Fairfield Way, 06269, CT, USA

\*Corresponding author. rye.howard-stone@uconn.edu

FOR PUBLISHER ONLY Received on Date Month Year; revised on Date Month Year; accepted on Date Month Year

## Abstract

**Summary:** We present AmpliconHunter2 (AHv2), a highly scalable *in silico* PCR engine written in C that can handle degenerate primers and uses a highly accurate melting temperature model. AHv2 implements a bit-mask IUPAC matcher with AVX2 SIMD acceleration, supports user-specified mismatches and 3' clamp constraints, calls amplicons in all four primer pair orientations (FR/RF/FF/RR), and optionally trims primers and extracts fixed-length flanking barcodes into FASTA headers. The pipeline packs FASTA into 2-bit batches, streams them in 16 MB chunks, writes amplicons to per-thread temp files and concatenates outputs, minimizing peak RSS during amplicon finding. We also summarize updates to the Python reference (AHv1.1).

**Availability and Implementation:** AmpliconHunter2 is available as a freely available webserver at: <https://ah2.uconn.engr.edu>. Source code is available at: <https://github.com/rhowardstone/AmpliconHunter2> under an MIT license. AHv2 was implemented in C; AHv1.1 using Python 3 with Hyperscan.

**Contact:** rye.howard-stone@uconn.edu

**Supplementary information:** Supplementary data are available at *Bioinformatics* online.

**Key words:** in silico PCR, AVX2, SIMD, IUPAC matching, barcodes, amplicon sequencing, streaming, 2-bit encoding

## Introduction

Polymerase chain reaction (PCR) amplicon sequencing remains a cornerstone of microbiome profiling because it is cost-effective, scalable and can generate taxonomic profiles from complex microbial communities. Most microbial surveys amplify portions of the 16S rRNA gene because this gene contains nine hypervariable regions (V1–V9) flanked by conserved segments that allow the same primers to amplify diverse bacterial taxa. Despite its ubiquity, amplicon sequencing can be affected by amplification bias: variability in primer binding sites across taxa means that some organisms may be under-represented in sequence data. Indeed, recent studies emphasize that commonly used universal primer sets often fail to capture the full microbial diversity in a sample and that primer design must account for inter-genomic variation (Sunthornthummas et al., 2025).

As *in vitro* experimentation is expensive, practitioners frequently use *in silico* tools to predict genome amplification patterns and assess off-target amplification before wet-lab experiments. Existing *in silico* tools, however, struggle to process currently available million-genome datasets or lack features such as the accurate modeling of melting temperature with mismatches required by commonly used sets of degenerate primers. To address these shortcomings, we recently released AmpliconHunter v1 (AHv1), a scalable *in silico* PCR package

implemented in Python (Howard-Stone and Măndoiu, 2026). AHv1 relies on the Hyperscan regex engine Wang et al. (2019) for performing approximate matching with mismatches and performs nearest-neighbor melting temperature calculations using the BioPython's Tm\_NN function.

In this paper we further improve the scalability of *in silico* PCR by introducing AmpliconHunter v2 (AHv2). AHv2 is written in C and implements a bit-mask IUPAC matcher with AVX2 SIMD acceleration. For this tool, we also implemented a C version of the nearest-neighbor melting temperature model that is identical to BioPython's Tm\_NN function. In the rest of the paper we describe AHv2 and provide benchmarking results comparing it with an updated version of AHv1 (AHv1.1) and several interim versions (Table 1).

## Implementations

**AHv1.1** is a functional update to AmpliconHunter (Howard-Stone and Măndoiu, 2026), that permits FASTQ input and output, optionally trims primer sequences, extracts fixed-length flanking barcodes and can automatically filter off-target amplicons. When published, AmpliconHunter was compared against prior efforts such as PrimerEvalPy (Vázquez-González et al., 2024) and Ribdif2 (Murphy and Strube, 2023), which offer utilities for *in silico* PCR, but are not designed for

million-genome scale. AHv1.1 retains nearly identical runtime characteristics as version 1, which was found to be 100 times faster than PrimerEvalPy and 10 times faster than Ribdif2, while supporting significantly more features.

**AHv2.α** was rewritten in C to use 2-bit compressed data for performance. Input FASTA files are compressed into 2-bit batches that store headers, lengths and packed bases. These batches are memory-mapped with `MAP_PRIVATE`; AHv2.α advises the kernel that pages will be accessed sequentially and dropped once consumed using `posix_madvise` with `POSIX_MADV_SEQUENTIAL` and `POSIX_MADV_DONTNEED`. The engine enforces a user-defined 3-prime clamp, streams amplicons to per-thread buffers and periodically merges them to minimize peak resident set size (RSS). AHv2.α outputs FASTA only and omits melting temperature ( $T_m$ ), HMM calculation, decoy analysis, and taxonomy modules for simplicity.

**AHv2.β** builds on AHv2.α by incorporating AVX2 parallelization, improved thread scheduling and dynamic buffer allocation. Primers are compiled into per-base IUPAC masks and matched using AVX2 256-bit registers; each register holds 32 bytes, allowing mismatches to be counted in parallel. AHv2.β achieves lower runtime but consumes untenable RAM.

**AHv2.γ** further optimizes the AVX2 matcher and I/O pipeline by reading data in 16 MB chunks. It hoists primer masks into contiguous vectors, employs unrolled loops to reduce branch mispredictions, and aggressively reclaims memory after each batch. AHv2.γ also uses static scheduling to mitigate load imbalance. These changes yield the fastest runtimes and the best parallel efficiency but still incur a moderate memory footprint due to decoded cache blocks.

**AHv2** reimplements BioPython’s entire `Tm.NN` function in C, permitting exact replication of the calculations in AHv1.1, for any parameter combination and all salt correction methods. Results are rounded to two decimal places but are otherwise identical to the BioPython annotations. AHv2 omits HMMs, decoys, taxonomy, and FASTQ features by design for minimal dependencies and maximal throughput.

## Results

We compared AHv1.1–AHv2 using six tests. All comparisons used 204.8K genomes from the AllTheBacteria genome collection, a set of 2.4M publicly available bacterial assemblies that we previously used for large-scale amplicon evaluation (Hunt et al., 2024). Instructions for downloading the genomes we tested are shown as part of our supplementary repository. Each test includes multiple replicates. The benchmarking script used the same set of primer pairs (V1V9) and parameters (two mismatches, clamp size of 3) across versions. Tests 1–5 were all conducted on the 12.8K genome subset. Filtering based on  $T_m$  is disabled in both implementations that support it: AHv1.1 and AHv2; however, both implementations automatically compute and annotate amplicons with  $T_m$  using the same nearest-neighbor model.

### Input size scaling

Figure 1A shows the runtime as the number of genomes increases (6.4K–204.8K). AHv2 consistently performs the fastest, followed by AHv2.γ, AHv2.β, AHv2.α and AHv1.1. AHv2 completes 204.8K genomes in 347.9 seconds (95% CI: [332.0, 363.8]), whereas AHv1.1 requires 2056.5 seconds (95% CI: [1925.5, 2187.6]).

| Implementation Detail           | v1.1 | v2.α | v2.β | v2.γ | v2 |
|---------------------------------|------|------|------|------|----|
| Python implementation           | ✓    | ✗    | ✗    | ✗    | ✗  |
| Hyperscan regex                 | ✓    | ✗    | ✗    | ✗    | ✗  |
| SIMD acceleration               | ✓*   | ✗    | ✓†   | ✓†   | ✓† |
| C implementation                | ✗    | ✓    | ✓    | ✓    | ✓  |
| 2-bit encoded pipeline          | ✗    | ✓    | ✓    | ✓    | ✓  |
| Memory-mapped I/O               | ✗    | ✓    | ✓    | ✗    | ✗  |
| Buffered I/O                    | ✓    | ✗    | ✗    | ✓    | ✓  |
| Bit-mask IUPAC                  | ✗    | ✓    | ✓    | ✓    | ✓  |
| Aggressive buffer reclamation   | ✗    | ✗    | ✗    | ✓    | ✓  |
| Melting Temperature calculation | ✓    | ✗    | ✗    | ✗    | ✓  |

**Table 1.** Binary feature and implementation matrix across AmpliconHunter versions (green check = supported, red X = not supported). Rows with identical values across all versions were removed. \*AHv1.1 utilizes Hyperscan which implicitly uses SIMD acceleration. † AmpliconHunter2 explicitly uses AVX2 instructions.

Memory usage (Figure S1A) tells a different story: AHv2.β uses significantly more RAM (45.8 GB at 204.8K genomes). AHv2 and AHv2.γ maintain moderate memory usage around 3.9 GB. AHv2.α uses ~0.73 GB, while AHv1.1 maintains the lowest RSS regardless of input size (0.48 GB for 204.8K genomes).

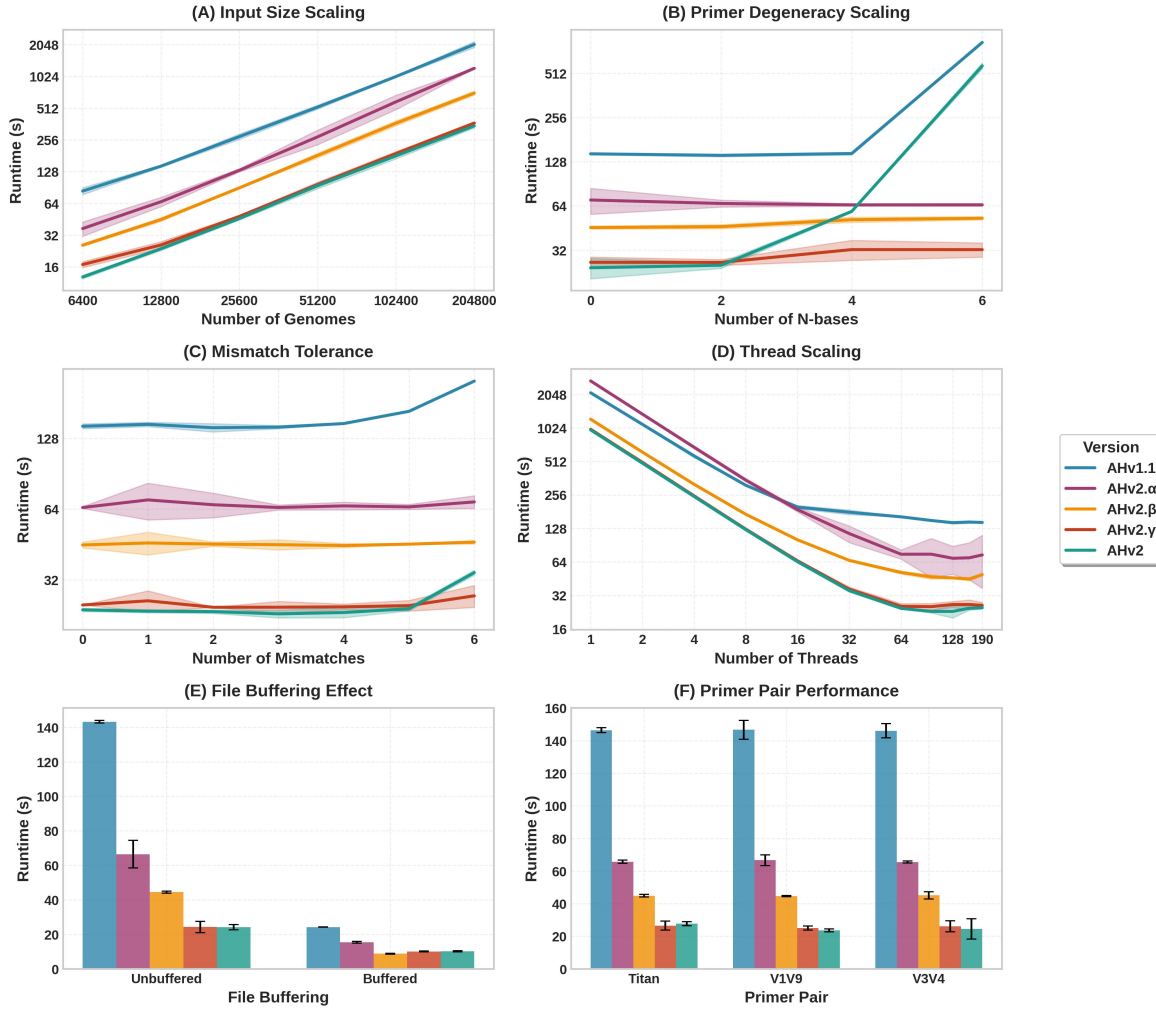
Scaling efficiency (Figure S1B) is computed as runtime for a baseline input divided by runtime for larger inputs normalized by input size. Perfect scaling would remain at 100%. AHv2.γ exceeds 140% efficiency at intermediate sizes. AHv1.1’s efficiency increases to ~130% at the largest input size, suggesting its better-than-linear scaling for the Python implementation is due to amortized overhead. AHv2 and AHv2.β peak just below 120%, while AHv2.α descends slightly beneath 100% at large inputs.

System-level metrics also reveal differences. All AHv2 variants read only a fraction of the input volume compared with AHv1.1 because they use compressed 2-bit batches, whereas AHv1.1 reads entire FASTA files (approximately 3.93× reduction in filesize). Context switching (Figure S1C) increases with the number of genomes, with AHv2.α showing the highest count at large genome sizes. AHv1.1’s system-time fraction (Figure S1D) decreases as the input grows (~7% at 6.4K genomes), while all C implementations maintain consistently low system overhead (1–4%).

### Primer degeneracy and mismatch tolerance

Degenerate primer bases (“N” positions) increase the size of the search space. In Figure 1B, two N bases are added at a time (one per primer) and used to extract amplicons from the 12.8K genome subset. AHv1.1’s runtime increases exponentially when primers contain six degenerate bases (reaching over 830 seconds), whereas the earlier C implementations maintain nearly constant runtime (AHv2.γ remains around 30 seconds). This highlights the advantage of bit-mask matching: AVX2 vectorizes IUPAC comparisons and avoids the combinatorial explosion inherent in regex matching. However, AHv2 also exhibits an exponential increase in response to increased degeneracy in input primers, as  $T_m$  needs to be calculated for more primer variants. Memory usage (Figure S2A) is largely unaffected by degeneracy for all implementations.

Mismatch tolerance has similar effects on the search space (Figure 1C). Allowing more mismatches slows all versions, but the C versions degrade more gracefully because mismatches can be counted efficiently using bitwise operations, whereas



**Fig. 1.** Runtimes for AHv1.1–AHv2 across six experiments. Lines are means; ribbons show 95% CI. (A) Input-size scaling across 6.4K–204.8K genomes. (B) Effect of primer degeneracy (number of appended N bases). (C) Effect of allowed mismatches (substitutions). (D) Thread scaling from 1–190 threads. (E) Cold vs warm cache performance. (F) Primer pair performance (Titan, V1V9, V3V4).

Hyperscan must evaluate many patterns. At six mismatches AHv1.1’s runtime increases to approximately 224 seconds, while AHv2 remains at 34.4 seconds. Memory footprints (Figure S2B) are largely flat, with a slight increase for AHv1.1 and AHv2.α at six allowed mismatches.

### Thread scaling and parallel efficiency

Figure 1D reports runtime as the number of threads increases (1–190). AHv2 achieves near-linear speed-ups up to 64 threads before saturating, completing the largest dataset in ~24.8 seconds at 190 threads. AHv2.γ scales similarly well, reaching about 26 seconds at 190 threads. AHv2.β scales well to 32 threads but plateaus thereafter just below 50 seconds. AHv2.α stabilizes around 74 seconds. AHv1.1 shows the poorest parallel scaling; beyond 16 threads additional cores provide diminishing returns, leveling off at ~144 seconds.

Parallel efficiency, the ratio of ideal to observed runtime scaling relative to a baseline input, is shown in Figure S2C. AHv2 and AHv2.γ maintain over 90% efficiency up to 16 threads and remain above 80% at 32 threads, but drop afterwards. AHv1.1 drops below 90% with as few as 8 threads,

plummeting to less than 40% efficiency with 32 threads. CPU utilization (Figure S2D) further illustrates these trends: AHv2, AHv2.γ and AHv2.β saturate cores, reaching over 4,000% utilization (about 40 cores × 100%), while AHv1.1 remains under 1,500% likely because Python’s Global Interpreter Lock serializes some operations.

### Cache performance

Figure 1E and Figure S2E evaluate cold versus warm cache conditions. When the OS page cache is cold (first run), AHv1.1 spends most of its time in disk I/O, taking ~140 seconds. AHv2.α completes in ~70 seconds, AHv2.β in ~45 seconds, AHv2.γ in ~25 seconds and AHv2 in 22 seconds. On repeated runs (warm caches), all implementations improve, but AHv2.β improves the most, becoming competitive with AHv2.γ and AHv2. The cache speed-up factor (Figure S2E) quantifies this improvement: AHv1.1 gains just shy of a 6× speed-up from caching, AHv2.α roughly 4.3×, AHv2.β about 5×, AHv2.γ approximately 2.4× and AHv2 about 2.3×.

## Primer-pair performance

We evaluated three commonly used primer pairs (Titan, V1V9 and V3V4) across all implementations (Figure 1F). For the Titan primer pair, AHv2 completes in 27.7 seconds, AHv2. $\gamma$  in 26.5 seconds, AHv2. $\beta$  in 44.9 seconds, AHv2. $\alpha$  in 65.7 seconds, and AHv1.1 in 146.4 seconds. Similar patterns hold for V1V9 and V3V4 primers with minimal variation between them. Memory usage was nearly identical between primer pairs for all methods. These results suggest that the level of degeneracy of primers and amplicon lengths (within practical ranges) have little effect on runtime beyond constant factors; performance differences are dominated by architecture and caching strategies.

## Discussion and future directions

Overall, our results highlight a trade-off between speed and memory. AHv2 is the fastest implementation, with a speed-up factor of  $5.91\times$  over AHv1.1 at 204.8K genomes (Figure S2F). Its optimizations and parallelizations make it the most suitable for large-scale projects where throughput is paramount. AHv2. $\gamma$  offers nearly equivalent speed ( $5.57\times$  speed-up) but does not include melting temperature calculation. AHv2. $\beta$  provides a  $2.89\times$  speed-up but requires substantial memory (45.8 GB). AHv2. $\alpha$  strikes a balance, delivering a mere  $1.68\times$  speed-up over AHv1.1 but maintaining a tiny memory footprint (0.73 GB). AHv1.1 remains useful for its advanced features (HMMs, taxonomic summarization, and FASTQ support) but is not competitive at scale.

The move from a Python + regex engine to a C + AVX2 bit-mask matcher yields dramatic performance gains. Vectorized matching allows 32 bases to be compared in a single instruction, and streaming 2-bit batches reduces I/O and memory traffic. The results show that naive regex matching struggles with degenerate primers and mismatch tolerance, whereas bit-mask approaches scale gracefully.

However, the AHv2 C implementation does not include all original functionality. It currently supports FASTA input only, omitting HMM scoring, decoy sequences, and taxonomic summaries; these features remain available in AHv1.1. AHv2 scales well to dozens of cores, but scaling saturates beyond 64 threads due to I/O bottlenecks and memory contention; further improvements could involve explicit NUMA-aware scheduling and prefetching (non-uniform memory access).

Future releases should aim to close the functionality gap by incorporating modules for HMM scoring and decoys while preserving speed. Support for FASTQ input with quality propagation would make the C implementation usable for processing amplicon sequencing data. Separating sequence headers as part of 2-bit compression may additionally shave some small amount of time off execution. Exploring AVX-512 and ARM NEON intrinsics could also provide speed-ups on newer hardware. In addition, GPU-accelerated matching may offer further performance gains.

## Conclusion

AmpliconHunter2 demonstrates that careful engineering, including vectorized IUPAC matching, 2-bit encoding, and careful memory management can significantly speed up demanding high-performance computing workflows. To increase the usability of AmpliconHunter2, we have released a corresponding webserver, as we did for AHv1. Users of the

webserver are empowered to assess the quality of their primers against large microbial genomic collections (including subsets of RefSeq (O’Leary et al., 2016) genomes and complete versions of the PATRIC Gillespie et al. (2011), GTDB (Parks et al., 2021), and AllTheBacteria databases (Hunt et al., 2024)), without access to command-line tools or high-performance computing environments. Our results show that the AHv2 webserver completes the analysis for V1V9 primers on the  $\sim 2.4$ M genomes from the AllTheBacteria project in 38.73 minutes, compared to 419.45 minutes for the AmpliconHunter webserver ( $\sim 10.8\times$  speedup). For reproducibility, we provide all intermediary implementations and benchmarking scripts on our supplementary GitHub page ([https://github.com/rhowardstone/AmpliconHunter2\\_benchmark](https://github.com/rhowardstone/AmpliconHunter2_benchmark)). Together, these implementations provide a flexible toolkit for microbiome researchers and primer designers operating at terabyte-scale.

## References

- J. J. Gillespie, A. R. Wattam, S. A. Cammer, et al. Patric: the comprehensive bacterial bioinformatics resource with a focus on human pathogenic species. *Infection and Immunity*, 79(11):4286–4298, 2011. doi: 10.1128/IAI.00207-11. URL <https://doi.org/10.1128/IAI.00207-11>.
- R. Howard-Stone and I. I. Măndoiu. Ampliconhunter: A scalable tool for pcr amplicon prediction from microbiome samples. In *Computational Advances in Bio and Medical Sciences (ICCABS 2025)*, volume 15599 of *Lecture Notes in Computer Science*, pages 329–344, Cham, 2026. Springer. doi: 10.1007/978-3-032-02489-3\_25.
- M. Hunt, L. Lima, W. Shen, et al. Allthebacteria: all bacterial genomes assembled, available and searchable. *bioRxiv*, pages 2024–03, 2024.
- R. Murphy and M. L. Strube. Ribdif2: expanding amplicon analysis to full genomes. *Bioinformatics Advances*, 3(1), 2023. doi: 10.1093/bioadv/vbad111. URL <https://doi.org/10.1093/bioadv/vbad111>.
- N. A. O’Leary, M. W. Wright, J. R. Brister, et al. Reference sequence (refseq) database at ncbi: current status, taxonomic expansion, and functional annotation. *Nucleic acids research*, 44(D1):D733–D745, 2016.
- D. H. Parks, M. Chuvpochina, C. Rinke, et al. Gtdb: an ongoing census of bacterial and archaeal diversity through a phylogenetically consistent, rank normalized and complete genome-based taxonomy. *Nucleic acids research*, 50(D1): D785–D794, 2021.
- S. Sunthornthummas, R. Wasitthankasem, P. Phokhaphan, et al. Unveiling the impact of 16s rna gene intergenomic variation on primer design and gut microbiome profiling. *Frontiers in Microbiology*, 16, 2025. doi: 10.3389/fmicb.2025.1573920. URL <https://doi.org/10.3389/fmicb.2025.1573920>.
- L. Vázquez-González, A. Regueira-Iglesias, C. Balsa-Castro, et al. Primerevalpy: a tool for in-silico evaluation of primers for targeting the microbiome. *BMC Bioinformatics*, 25, 2024. doi: 10.1186/s12859-024-05805-7. URL <https://doi.org/10.1186/s12859-024-05805-7>.
- X. Wang, Y. Hong, H. Chang, et al. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 631–648, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2.

## Supplementary Information

### Benchmarking script and reproducibility

The benchmarking script (`run_benchmarks.sh`) iterates over the six tests shown in 1. For each test, it runs all four versions of AmpliconHunter using comparable parameters, captures wall-clock time with `/usr/bin/time -v` and collects system metrics via `/proc/self/status` and `perf stat`. The script writes a JSON summary (`benchmark_results.json`) that includes runtimes, peak RSS, page-fault counts, context switches, and CPU utilization for each replicate. Users can reproduce the plots by running `generate_publication_figures.py` on the JSON file. The repository also contains example primer files and a manifest of compressed genome batches.

### Hardware and software environment

Benchmarks were performed on an Ubuntu 22.04 server with dual Intel Xeon Gold 6248R CPUs (2×24 cores, 3.0 GHz) and 512 GB of DDR4 memory. The C implementations were compiled with GCC 13.3 using `-O3`, `-march=native` and `-mavx2`. AHv1.1 used Python 3.11 and Hyperscan 0.7.8.

### Data availability

All code, compressed genome batches, benchmarking scripts and raw results are available from the accompanying GitHub repository ([https://github.com/rhowardstone/AmpliconHunter2\\_benchmark](https://github.com/rhowardstone/AmpliconHunter2_benchmark)). Figures were generated using the provided scripts and can be reproduced on any modern Linux server. The authors welcome contributions and feature requests.

### Webserver

For ease of use, we make a webserver available serving AmpliconHunter2 on the same databases used for the original AmpliconHunter web interface, with a slightly improved design. Past jobs are now much easier to find: searchable, and filterable by database as well as status. Plots are largely the same, with the removal of the HMM and decoy plots, but we have added a taxonomy breakdown that is searchable as a table, and navigable as a tree. We have kept our amplitype pattern plots, but opted for a static png for easy transferability. We have included several interactive plots, including distributions for amplicon length, GC content, and melting temperature, along with the primer orientation breakdown. Please visit <https://ah2.engr.uconn.edu/> to view and submit AmpliconHunter2 jobs.

### Primer matching and clamp logic (AHv2)

Primers are converted to per-base IUPAC bit masks; reverse-complement masks are precomputed. Each sequence is converted once to a mask array. We count mismatches via AVX2 bitwise AND operations, with exact 3' clamp enforced.

### Amplicon calling and orientation (AHv2)

We sort candidate sites and pair opposite-sense hits within user bounds (`--min-length`, `--max-length`). We stop when a same-sense site appears (prevents invalid overlaps). Orientation codes are the same as version 1: FR, RF, FF, RR. By default, we emit FR+RF; `--include-offtarget` additionally emits FF/RR. RF amplicons are reverse-complemented so sequences are in forward orientation.

### Primer trimming and barcode extraction (AHv2)

With `--trim-primers`, we remove matched primer sequences from the emitted amplicon. Barcodes are fixed-length flanks upstream of the forward primer (`--fb-len`) and downstream of the reverse primer (`--rb-len`) for FR; the RF case extracts on the opposite sides and reverse-complements both barcodes.

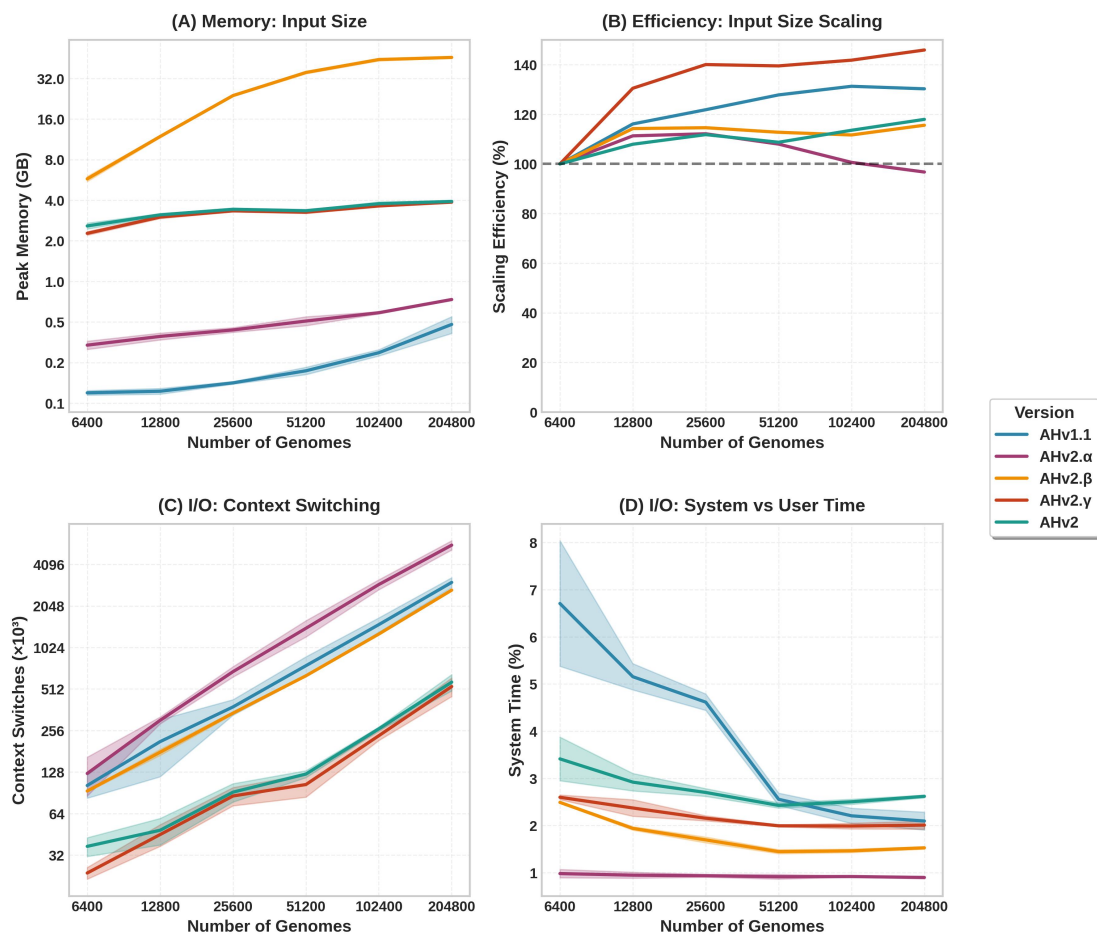
### Headers and outputs (AHv2)

Output is FASTA. Headers encode source file, genomic coordinates, orientation, matched primer snippets, and optional barcodes, e.g.:

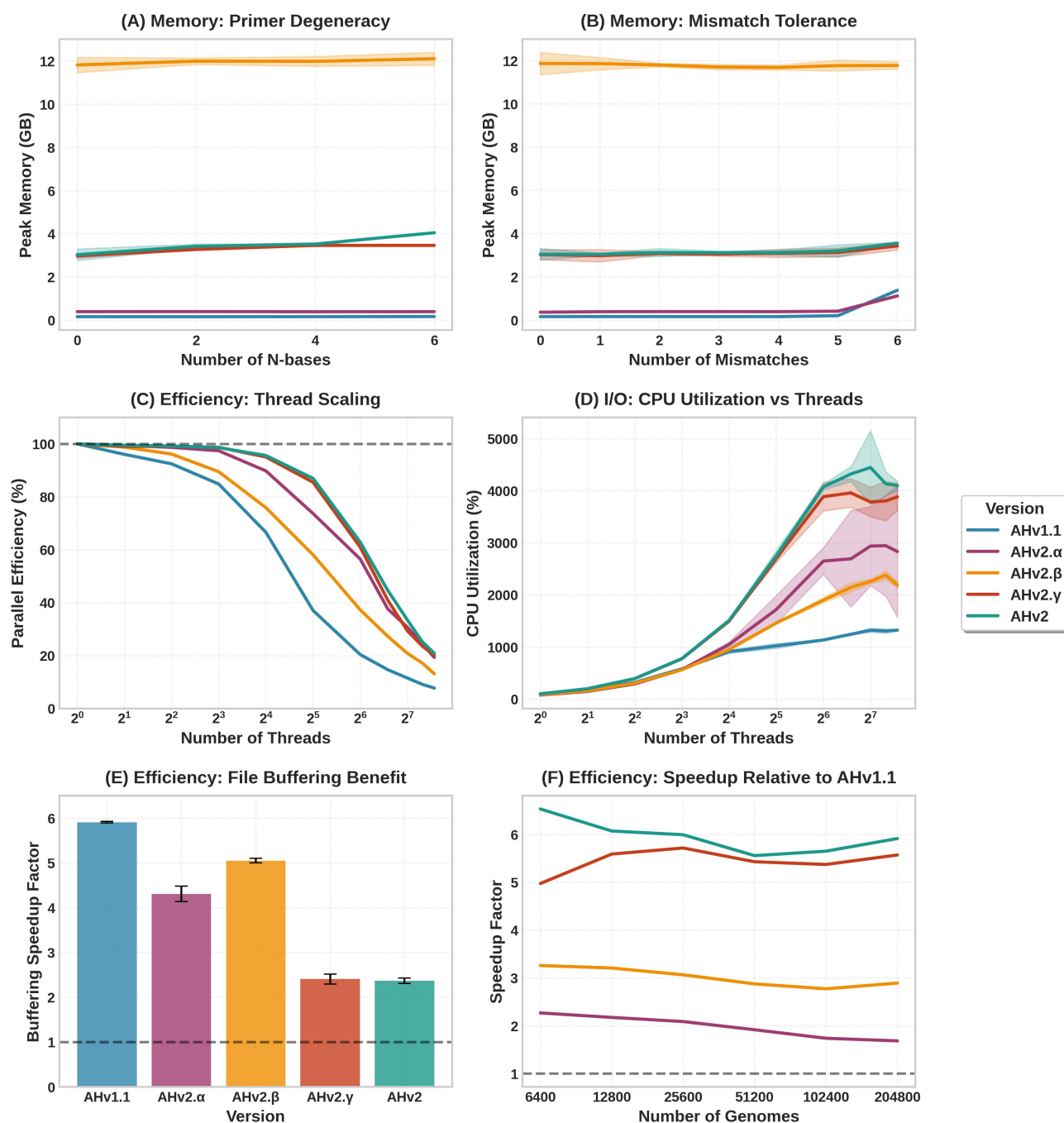
```
>seqid.source=GCF_XXXX.fa.coordinates=12345-13567.Tm=60.42.orientation=FR
.fprimer=... .rprimer=... .fb=ACGT ... .rb=TGCA
```

(.fb/.rb only when requested.)

## Supplementary Figures



**Fig. S1.** Memory and efficiency analysis for input size scaling. Panel (A) shows peak resident memory (GB) versus input size, (B) input size scaling efficiency, (C) context switching overhead, and (D) ratio of system time to user time. Means with replicate variability shown (95% CI).



**Fig. S2.** Extended performance analysis. Panel (A) shows peak memory for primer degeneracy, (B) peak memory for mismatch tolerance, (C) parallel efficiency versus thread count, (D) CPU utilization versus thread count, (E) cache performance benefit, and (F) speed-up relative to AHv1.1. Means with replicate variability shown (95% CI).