GCN10: An MPI-parallelized framework for processing global curve number rasters for hydrologic modeling*

Muhammad Abdullah Azzam[a], Huidae Cho[a,*]

[a]*Department of Civil and Environmental Engineering, New Mexico State University, Las Cruces, NM 88003, USA*

Abstract

GCN10 is an open, MPI-parallelized C program generating 10 m curve number rasters from ESA WorldCover 2021 landcover and HYSOGs250m hydrologic soil group datasets for various hydrologic and antecedent runoff conditions. It enforces cell alignment, uses windowed GDAL I/O, and writes Cloud Optimized GeoTIFFs. Outputs deliver high-resolution curve number inputs for flood screening, watershed planning, and hydrologic modeling in data-scarce regions. A thin Python wrapper gcn10py enables scripting and integration into model-preparation workflows. Benchmarks scale a 32-block test from 7866 s (1 rank) to 845 s (16 ranks). Global production completes in approximately 12 h 48 min on one node with an i9-13900KS CPU, enabling ensembles of hydrologic and antecedent runoff conditions.

Keywords: Curve number, Hydrology, Raster processing, MPI parallelization

CITATION: Azzam, M. A., Cho, H., Accepted in May 2026. GCN10: An MPI-parallelized framework for processing global curve number rasters for hydrologic modeling. SoftwareX, 102725.

*Corresponding author.
Email address: hcho@nmsu.edu (Huidae Cho)

Metadata

Code metadata

Nr.	Code metadata description	Metadata
C1	Current code version	0.1.0
C2	Permanent link to code/repository used for this code version	https://github.com/clawrim/gcn10
C3	Permanent link to Reproducible Capsule	https://github.com/clawrim/gcn10
C4	Legal Code License	GNU General Public License (GPL) v3
C5	Code versioning system used	Git
C6	Software code languages, tools, and services used	C, CMake, MPI
C7	Compilation requirements, operating environments & dependencies	C Compiler, CMake, MPI, GDAL
C8	If available Link to developer documentation/manual	https://github.com/clawrim/gcn10/blob/main/README.md
C9	Support email for questions	mabdazzam@outlook.com

1. Motivation and significance

The Curve Number (CN) method is a standard loss formulation used in design hydrology, flood management, and watershed planning across various agencies and practices [1, 2]. It translates Land Use and Land Cover (LULC), Hydrologic Soil Group (HSG), Hydrologic Condition (HC), and Antecedent Runoff Condition (ARC) into a single parameter that governs abstractions before surface runoff discharge. Its simplicity enables rapid screening and model setup, yet it is empirical, sensitive to input classification and scale, and can perform poorly for extremes or heterogeneous terrains without calibration [3]. Operational hydrology increasingly requires consistent, high-resolution CN maps derived from contemporary sources such as the Global Hydrologic Soil Groups (HYSOGs250m) HSG dataset at 250 m rather than coarse, static lookups [4]. Recent evidence suggests that finer spatial detail and dynamic representation enhance event-based runoff prediction skills at regional scales [5]. For the Continental United States, a 30 m time-varying CN dataset increased the Nash-Sutcliffe Efficiency (NSE) from 0.68 to 0.71 across 230 events and reduced the Root Mean Square Error (RMSE) during extremes from 63.96 mm to 54.24 mm compared to a 250 m baseline, with gains attributed to finer LULC and HSG detail and slope adjustment [5]. Higher-resolution LULC has also improved Soil and Water Assessment Tool (SWAT) [6] streamflow metrics by 1 % to 14 % in high-elevation basins, reinforcing the role of spatial detail in loss characterization [7]. Producing high-resolution CN rasters across large watersheds is dominated by geospatial integration and Input/Output (I/O), and ad hoc desktop chains struggle to scale. As a result, many projects downscale coarsely, reuse legacy layers, or avoid ensembles across HCs and ARCs despite their implications for design flows. There is a need for a rigorously engineered and reproducible path from modern land cover and soil rasters to high-resolution CN products that integrate cleanly with mainstream hydrologic

Table 1: Default lookup CSV files distributed with GCN10. File names follow the pattern `default_lookup_<hc>_<arc>.csv`, where `f`, `g`, and `p` denote fair, good, and poor hydrologic condition, respectively, and `i`, `ii`, and `iii` denote ARC-I, ARC-II, and ARC-III.

CSV file	**Hydrologic condition**	**ARC**
`default_lookup_f_i.csv`	fair	I
`default_lookup_f_ii.csv`	fair	II
`default_lookup_f_iii.csv`	fair	III
`default_lookup_g_i.csv`	good	I
`default_lookup_g_ii.csv`	good	II
`default_lookup_g_iii.csv`	good	III
`default_lookup_p_i.csv`	poor	I
`default_lookup_p_ii.csv`	poor	II
`default_lookup_p_iii.csv`	poor	III

toolchains. The 10 m target resolution in GCN10 follows the native ESA WorldCover grid, while the effective soil information content remains bounded by the coarser HYSOGs source that is resampled to that grid during preprocessing.

Existing tools such as GRASS [8] addon r.curvenumber [9], QGIS CN Generator plugin [10], ArcGIS Pro SCS CN toolbox, and Google Earth Engine (GEE) [11] applications for 250 m CN visualization are valuable for local analyses, yet they are slow or brittle at continental scales, tied to desktop Geographic Information System (GIS) stacks, or lack offline processing with deterministic tiling and resume safety. The Global Curve Number 10m (GCN10) dataset addresses these gaps with a compiled, Message Passing Interface (MPI) [12]-parallelized C program that computes CN at 10 m from the European Space Agency (ESA) WorldCover 2021 [13] and HYSOGs datasets using a dual HSG lookup and user-selected hydrologic modifiers. This dual-HSG option allows drainage-dependent interpretation when soils are represented by dual classes in the source data, so drained and undrained assumptions can be evaluated explicitly within the same workflow. When source rasters differ in native resolution, GCN10 aligns them to a common grid defined by the selected reference raster. In this study, the 250 m HYSOGs dataset was reprojected and resampled to the 10 m ESA WorldCover grid to enable cell-by-cell CN lookup, while recognizing that the effective soil information content remains constrained by the native coarser source resolution. The default lookup tables distributed with GCN10 encode the Curve Number assignments used by the processing framework. The ARC-II values were compiled from the USDA NRCS National Engineering Handbook (NEH), Part 630, Chapter 9 [2], and the ARC-I and ARC-III values were assigned using the tabulated conversions in Chapter 10 [14]. These lookup tables are shipped with the software in the `lookups/` directory of the source tree (for example, `~/usr/local/src/gcn10/lookups` in the Linux workflow described in Section 2.1 of the documentation), so users can inspect, edit, or replace the default values directly to support alternative CN conventions, local calibration practice, or region-specific adaptations. Table 1 lists the default lookup CSV files currently provided with GCN10. Users may modify any of these CSV files before execution to change the default CN assignments used by the compiled backend without altering the source code itself.

At a high level, GCN10 takes modern global land-cover and soil products as inputs and

produces tiled, analysis-ready CN rasters for multiple hydrologic conditions, antecedent runoff conditions, and drainage assumptions. Its Python wrapper gcn10py exposes the compiled backend for direct use in scripts and model-preparation pipelines. Section 2 describes the architecture, data flow, and parallelization strategy in detail. This wrapper eliminates many hand-crafted GIS steps (for example, manual clipping, misaligned mosaics, and inconsistent NoData handling), converts high-resolution CN generation into a routine preprocessing step, and enables ensembles that were previously impractical. Downstream, the rasters integrate directly with loss methods in the Hydrologic Engineering Center Hydrologic Modeling System (HEC-HMS) and comparable rainfall-runoff models, simplifying calibration, subbasin attribution, and sensitivity analyses [15, 6].

Beyond delivering software, this paper contributes an evaluation of node-level parallelization strategies for embarrassingly parallel geoprocessing in a process-based, map-style workload with no spatial coupling. We implement and benchmark Python multiprocessing, a C hybrid MPI+OpenMP variant [12, 16], and a C pure MPI variant on identical hardware and inputs, and we report wall time, speedup, and efficiency. Results favor a Single Program, Multiple Data (SPMD), block-streaming MPI design that minimizes communication and preserves locality, which is consistent with Amdahl and Gustafson expectations for low communication transforms [17, 18]. These findings generalize to stencil-free raster parameter pipelines where computation per cell is light, windowed I/O dominates, and reproducibility depends on deterministic tiling and writers. In summary, GCN10 enables high-resolution CN generation to be fast, reproducible, and easy to automate, while the parallelization assessment helps practitioners choose effective node-scale execution models.

Finally, this capability supports data-scarce regions and large-scale studies by operationalizing high-resolution CN generation at practical runtimes. It aligns with open science practices by promoting transparent provenance and repeatable outputs, and it lowers the barrier to seasonal or scenario-based CN analyses that connect to flood risk assessment and water management objectives.

This paper presents GCN10 as a software framework for scalable, reproducible generation of high-resolution Curve Number rasters from large geospatial datasets. The main contribution is the design and implementation of the MPI-parallelized processing workflow, including its lookup-based CN assignment, deterministic tiling strategy, and computational performance. The objective of this study is therefore software development and evaluation in terms of functionality, reproducibility, and scalability. Evaluation of downstream hydrologic-model performance in rainfall–runoff applications such as HEC-HMS, SWAT, or similar systems is beyond the scope of the present manuscript because such applications depend on additional modeling choices including rainfall inputs, basin discretization, routing formulation, and calibration.

2. Software description

GCN10 is a compiled, MPI-parallel C program that computes CN at 10 m resolution from ESA WorldCover land covers and HYSOGs hydrologic soil groups using a dual HSG lookup with user-selected hydrologic modifiers. The workflow enforces cell alignment, uses windowed GDAL I/O, and writes tiled and compressed COGs with a fixed NoData value. A thin Python wrapper gcn10py exposes the compiled backend so users can call

the same high-performance C engine from scripts and integrate it into model-preparation pipelines (see *Python wrapper* on page 10).

The workload is embarrassingly parallel: each output block depends only on the ESA and HYSOGs pixels within that block, with no cross-block dependencies or halos, allowing blocks to be computed independently. This pattern applies a single per-block CN kernel independently across many tiles, naturally motivating an SPMD MPI design. The kernel reads ESA and HYSOGs windows, performs the dual HSG lookup, and writes one CN tile (see Section 2.2).

2.1. Assessment of parallelization techniques

We implemented four variants to identify a fast and robust strategy for large CN production. The variants were Python serial, Python `multiprocessing.Pool`, C hybrid MPI+OpenMP, and C MPI-only. Because the workload is embarrassingly parallel and has no cross-block coupling, an SPMD design is natural. Hybrid threading shortened the cell loop but did not improve end-to-end throughput, as the wall time was dominated by windowed reads, alignment, and serialized raster writes. GDAL's raster *write* paths are not thread-safe across threads, which further limits intra-process speedups on this I/O-bound pipeline [19]. Pure MPI kept more independent I/O and writes in flight at the process level with negligible communication, simpler control, and deterministic outputs. We therefore adopt pure MPI for production.

Benchmarking. All tests used identical inputs and hardware (Table 2). We conducted a 32-block benchmark test that consists of 32 raster blocks spanning Utah, Nevada, Colorado, New Mexico, Arizona, and Texas. Each block is a cell-aligned window into ESA WorldCover (10 m) and HYSOGs (250 m) resampled by nearest neighbor to the target grid. Inputs were read as Virtual Rasters (VRTs) [20] and GeoTIFF from disk, and outputs were written as tiled COG GeoTIFFs with Lempel–Ziv–Welch (LZW) compression and a fixed NoData value of 255. Performance was measured with the GNU time command (`/usr/bin/time`) [21]. Unless stated otherwise, the single-rank execution for C MPI and hybrid implementations is the *baseline* ($p = 1$). We report parallel speedup and efficiency as shown in (1) and (2) respectively:

$$S_p = \frac{T_1}{T_p} \tag{1}$$

and

$$E_p = \frac{S_p}{p} = \frac{T_1}{p\,T_p} \tag{2}$$

where T_1 denotes the baseline wall time for the serial implementation on the same hardware and inputs, T_p is the wall time when using p identical processing elements on the same problem size (strong scaling), p is the number of processing elements (e.g., MPI ranks, workers, or threads), S_p is the speedup at p, and E_p is the corresponding parallel efficiency (dimensionless, with $0 \leq E_p \leq 1$).

Performance evaluation. Table 3 reports measured mean wall times for the 32-block benchmark. The single-rank C MPI baseline completed the workload in 7866 s (average over 30 runs), and the 16-rank C MPI run completed it in 845 s (average over 30 runs).

Table 2: Hardware and system specifications for performance testing.

Component	**Specification**
Architecture	x86_64 (64-bit)
CPU model	Intel Core i9-13900KS (13th Gen)
Memory	128 GiB
Physical cores	24
Logical CPUs	32
Operating System	Linux Kernel 6.10.7
Compiler collection	GCC 14.2.0
MPI compiler	Open MPI 4.1.6
GDAL	3.11.0

Table 3: Measured mean wall times for the 32-block benchmark. C times are means over 30 runs, Python pool is a mean over 5 runs.

Implementation (32 blocks)	**Mean wall time**
C MPI (1 rank)	7866 s
C MPI (16 ranks)	845 s
Python `multiprocessing.Pool` (16 workers)	350 min

The rank-wise wall-time trend in Figure 2 shows the largest reduction from one to eight MPI ranks, followed by diminishing returns as raster reads and serialized writes become dominant. A 16-worker Python `multiprocessing.Pool` required 350 min for the same workload (average over 5 runs). Table 4 reports extrapolated times to generate the full global dataset under identical I/O conditions: C MPI (production run) completes global generation in 12 h 48 min, whereas the 16-worker Python pool requires 20 d and a serial Python upper-bound estimate requires 322 d. Hybrid C MPI+OpenMP matched pure MPI on wall time for the 32-block workload, but added scheduling overhead without end-to-end gains because of I/O dominance and thread-unsafe writers [19] as shown in Figure 3. The efficiency curve (Figure 1) matches the expected taper from Amdahl and Gustafson analyses once serialized I/O dominates. Global production runs were also executed on the NMSU Discovery cluster [22] with identical code and configuration. These results support a pure SPMD MPI design for this workload.

2.2. Software architecture

The design combines static block decomposition with rank-local kernel execution, where each MPI rank runs the same CN kernel on its own subset of blocks without exchanging data with other ranks. This design yields high throughput, consistent load balance, and reproducible outputs on single nodes and small clusters. The program runs SPMD on a single flat communicator (`MPI_COMM_WORLD`) created by the launcher with p ranks (the MPI size). Each rank owns a cyclic share of blocks via `for (i = rank; i < n_blocks; i += p)`, while rank 0 also performs lightweight logging and progress polling to avoid any central bottleneck. A final `MPI_Barrier` and collective finalize ensure clean

Table 4: Wall clock times to generate the global dataset based on the 32-block measurements and identical I/O conditions, and the Python parallel and serial times are indicative estimates derived from the 16-worker pool assuming ideal scaling.

Method (global dataset)	**Estimated total time**
C MPI (production run)	12 h 48 min
Python `multiprocessing.Pool` (16 workers)	20 d
Python serial (estimated)	322 d

termination with natural load balance and no master idle time. The whole workflow is summarized in Figure 4.

Rank 0 additionally tracks global progress. After it completes the kernel for one block, it increments an internal counter and drains nonblocking PROG ticks sent by worker ranks. This design keeps progress reporting lightweight and avoids introducing a central bottleneck or extra synchronization into the CN computation itself.

Algorithm 1 shows the main program. The manager maintains the queue and issues tasks, while workers run the CN kernel 2 and report completion or failure. Tagged messages (`WORK`, `DONE`, `STOP`, `FAIL`) provide simple flow control and fault handling. The CN kernel performs all raster operations locally within each rank. Algorithm 2 shows the kernel computation. It loads land-cover and soil windows, verifies coordinate systems, resamples soil data using the nearest neighbor method, and applies the dual HSG lookup for each cell, utilizing the specified drainage and hydrologic settings. A new raster window is written as a tiled and compressed COGs. No global communication is required during computation.

2.3. Software functionalities

GCN10 validates inputs, aligns grids, builds VRTs, and writes a deterministic manifest. It generates CN rasters by distributing blocks in a static round-robin pattern across MPI ranks (Algorithm 1) and executing a rank-local kernel for each block (Algorithm 2). The backend reads ESA windows, resamples HYSOGs to the target grid, applies the dual HSG lookup, and writes COGs with a fixed NoData value [23, 24, 25].

Input sources can be local rasters or virtual mosaics described by GDAL VRTs. Cell alignment is enforced at read time with explicit Coordinate Reference System (CRS) checks. Hydrologic condition, antecedent runoff condition, and drainage mode are selected for each run and combined into a Cartesian product of variants. Per-block compute is independent, enabling elastic scaling across MPI ranks. Outputs are tiled and compressed COGs with a stable NoData value for consistent downstream masking. Filenames and counts are invariant to the number of ranks. Existing outputs are detected and can be skipped or overwritten via a flag. Interrupted runs can be resumed without needing to rename or rewrite completed files. Each run writes concise progress messages and per-block timings to log files. Failures are recorded with explicit status codes and block IDs in the manifest, so problematic tiles can be re-run or inspected without regenerating the full dataset. This design allows reproducible and resumable runs.

A minimal text configuration specifies inputs and log directories. An optional text file containing the block IDs can also be passed to generate the dataset for those blocks

Require: $\mathbf{C}$ ▷ config (paths only: lc, hsg, lut, logdir)
Require: $\mathbf{R}_{\text{lc}}$ ▷ land-cover source
Require: $\mathbf{R}_{\text{hsg}}$ ▷ HSG source
Require: $\mathbf{L}$ ▷ CN lookup
optional: $\mathbf{B}_0$ ▷ block-id list; else enumerate all
internal: $D = \{d, ud\},\ H = \{p, f, g\},\ A = \{i, ii, iii\}$ ▷ kernel conditions

`MPI_Init`()
$(r, p) \leftarrow$ (`rank`, `size`); `logger`$\leftarrow$ `init`($\mathbf{C}$.`logdir`)
$B \leftarrow \begin{cases} \mathbf{B}_0, & \text{if provided} \\ \textsc{EnumerateBlocks}, & \text{otherwise} \end{cases}$; $n \leftarrow |B|$
fix bijection $\tau_B : \{0, \ldots, n-1\} \to B$ ▷ deterministic block order
$P_r \leftarrow \{\, j \in [0..n-1] \mid j \equiv r \pmod{p} \,\}$ ▷ round-robin over blocks
$S \leftarrow |D| \cdot |H| \cdot |A|$ ▷ conditions per block (kernel-internal)
if $r = 0$ **then**
    $prog\_expected \leftarrow S \cdot \big(n - \textsc{CountRR}(0, p, n)\big)$; $prog\_done \leftarrow 0$ ▷ expected worker ticks; init
    `MPI_Irecv`(m, src = `MPI_ANY_SOURCE`, tag = PROG, q) ▷ post rolling recv
end if
for all $j \in P_r$ **do**
    $b \leftarrow \tau_B(j)$
    $\mathrm{K}(\mathbf{R}_{\text{lc}}, \mathbf{R}_{\text{hsg}}, \mathbf{L}, b, r)$ ▷ Algorithm 2; emits worker ticks internally
    if $r = 0$ **then**
        $prog\_done \leftarrow prog\_done + S$ ▷ manager accounts for its own block
        while `MPI_Test`(q, *flag*) **and** *flag* **do**
            $prog\_done \leftarrow prog\_done + 1$; $\textsc{publish\_progress}(prog\_done/(S \cdot n))$ ▷ drain and report
            `MPI_Irecv`(m, `MPI_ANY_SOURCE`, PROG, q) ▷ keep one posted
        end while
    end if
end for
if $r = 0$ **then** ▷ drain to threshold
    while $prog\_done < prog\_expected$ **do**
        while `MPI_Test`(q, *flag*) **and** *flag* **do**
            $prog\_done \leftarrow prog\_done + 1$; $\textsc{publish\_progress}(prog\_done/(S \cdot n))$
            `MPI_Irecv`(m, `MPI_ANY_SOURCE`, PROG, q)
        end while
    end while
    `MPI_Cancel`(q); `MPI_Request_free`(q)
end if
`MPI_Barrier`()
`MPI_Finalize`()

Algorithm 1: MPI orchestrator. Blocks B are assigned round-robin via τ_B; every rank computes its subset P_r locally using the CN kernel (Algorithm 2). Rank 0 additionally tracks progress by tallying PROG ticks sent from workers while accounting for its own blocks directly.

Require: $\mathbf{R}$ ▷ open handles: lc, hsg
Require: $\mathbf{L}$ ▷ lookup; $[0, 100] \cup \{255\}$
Require: b ▷ block id
Require: ρ ▷ mpi rank

function $K(\mathbf{R}, \mathbf{L}, b, \rho)$
  $B \leftarrow$ BBOX(block b)
  $(L_c, n_x, n_y, gt, crs, nd_{lc}) \leftarrow$ READLC(B)
  $(H_c, s_x, s_y, sgt, scrs, nd_{hsg}) \leftarrow$ READHSG(B); **assert** $crs = scrs$ ▷ CRS guard; no reprojection
  $H \leftarrow$ UPSAMPLE_NEAREST(H_c; s_x, s_y, $sgt \to n_x, n_y$, gt) ▷ cell-center mapping; clamped indices
  ND $\leftarrow 255$ ▷ global NoData sentinel
  $\phi_d : A/D, B/D, C/D \mapsto \begin{cases} D, & d = \texttt{d} \\ A, B, C, & d = \texttt{ud} \end{cases}$ ▷ dual-HSG disambiguation
  for all $d \in \{d, ud\}$ **do**
    for all $h \in \{p, f, g\}$ **do**
      for all $a \in \{i, ii, iii\}$ **do**
        $K_r \leftarrow$ byte raster $(n_x \times n_y)$ filled with ND
        for $r \leftarrow 0$ **to** $n_y - 1$ **do**
          for $c \leftarrow 0$ **to** $n_x - 1$ **do**
            $lc \leftarrow L_c[r, c]$; **if** $lc = nd_{lc}$ **then continue** ▷ NA propagates
            $sg \leftarrow \phi_d(H[r, c])$; **if** $sg = nd_{hsg}$ **then continue**
            $v \leftarrow \mathbf{L}[d][h][a][sg][lc]$; **if** $v <$ ND **then** $K_r[r, c] \leftarrow v$
          end for
        end for
        $\omega \leftarrow$ writer opts ▷ tiled, lzw, nodata=ND (deterministic)
        $tmp \leftarrow$ PATH$(b, h, a, d) +$ ‘‘`.tmp`’’; $dst \leftarrow$ PATH(b, h, a, d)
        WRITE_COG(tmp, K_r, gt, crs, ω); `FLUSH+RENAME`$(tmp \to dst)$
        if $\rho \neq 0$ **then** ▷ unit tick (workers only)
          `MPI_Isend`(1, dest = 0, tag = PROG, s)
          `MPI_Request_free`(s)
        end if
      end for
    end for
  end for
end function

Algorithm 2: CN kernel (per block). Reads LC/HSG for block b, upsamples HSG to LC grid, and for all (d, h, a) writes a deterministic COG (tiled, LZW, NoData=255). After each (d, h, a) completion, workers (ranks $\rho \neq 0$) send one nonblocking `PROG` tick; the manager does not self-send.

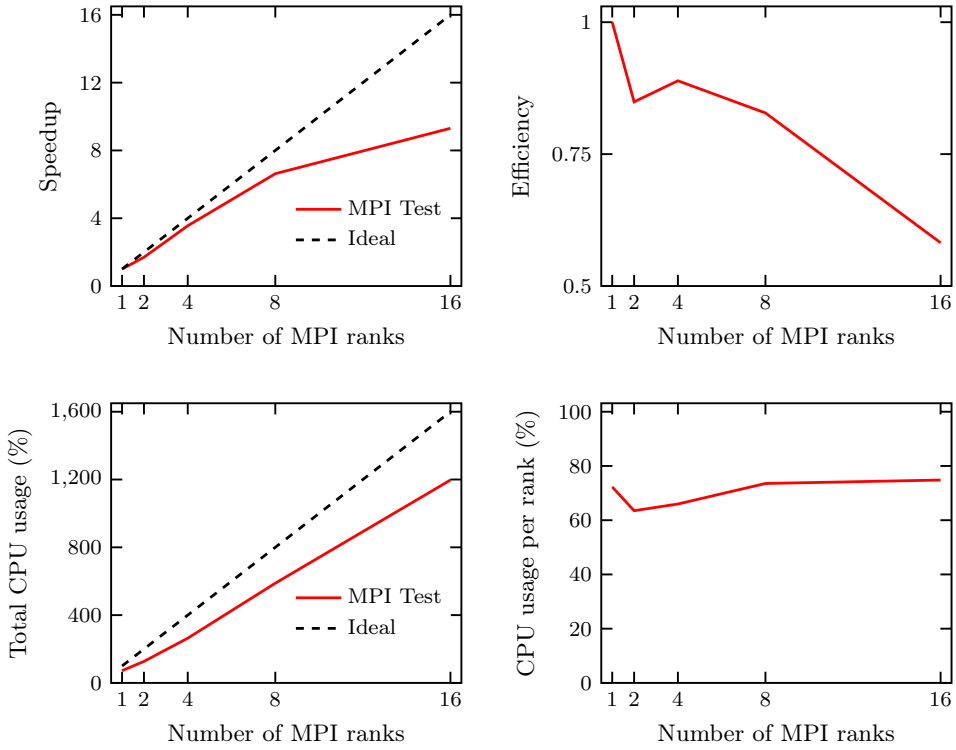


Figure 1: C MPI scales near linearly to eight ranks with efficiency exceeding 70 % before tapering due to I/O limits.

only. Output directories are created in the current working directory.

Command-line interface. The primary command-line flags are summarized in Table 5, and the examples below show common single-rank, block-restricted, and MPI executions.

```
# compiled binary (single rank)
gcn10 -c config.txt

# run specific blocks with overwrite flag
gcn10 -c config.txt -l blocks.txt -o

# MPI (example with Open MPI)
mpirun -n 16 gcn10 -c config.txt -l blocks.txt
```

Python wrapper (gcn10py). The gcn10py package [26] exposes the compiled backend through a single function `run_gcn10(args=None)` that locates the packaged `gcn10` executable and forwards a list of command-line arguments to it. When `args` is `None`, it defaults to `sys.argv[1:]` so driver scripts can pass their own command-line options unchanged. Higher-level helpers (for example, a function `run_gcn10(cfg, block_ids)`

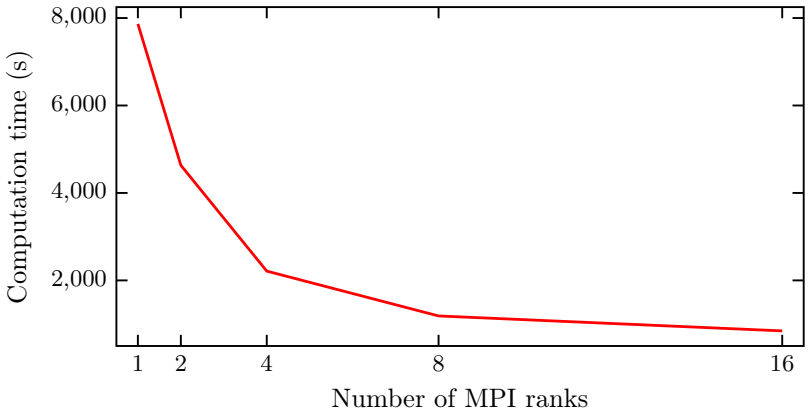


Figure 2: Wall time falls steeply from one to eight ranks and then flattens as windowed reads and serialized writers dominate.

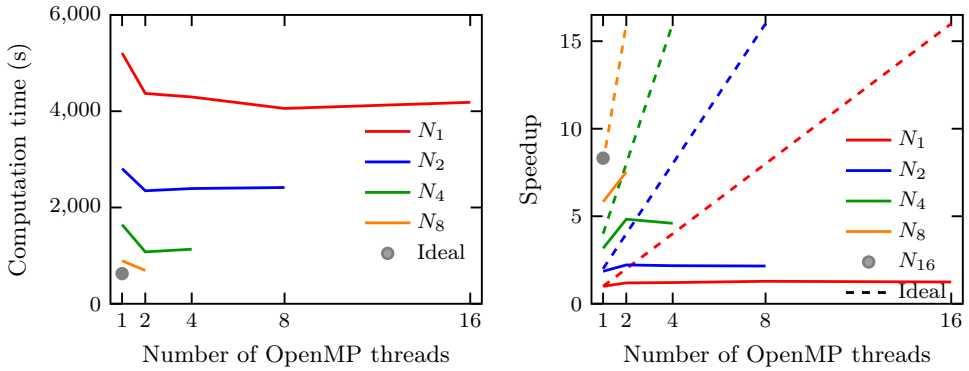


Figure 3: Hybrid MPI+OpenMP reduces per-cell compute time but yields little or no end-to-end gain once I/O and writers dominate.

in the testing scripts) simply assemble the arguments (configuration file, block list, overwrite flag) and then delegate to `run_gcn10`. All CN computation, windowed raster I/O, and MPI parallelism remain in the C code. The wrapper introduces only process-launch and argument-parsing overhead, which is negligible compared to multi-minute or multi-hour runs. It also provides utilities to select block IDs by spatial intersection with the user's Area of Interest (AOI), to perform simple mosaics, and to clip variants to vector boundaries. The wrapper preserves the same deterministic outputs as the binary and does not alter the MPI execution model. This way, GCN10 can be integrated with any workflow that leverages established spatial analysis libraries (e.g., GDAL, Rasterio, NumPy, Pandas, etc.) and provides a high-performance, streamlined workflow. Therefore, downstream tasks such as mosaicking, clipping, and zonal statistics can be executed with standard GDAL utilities or external hydrologic toolchains.

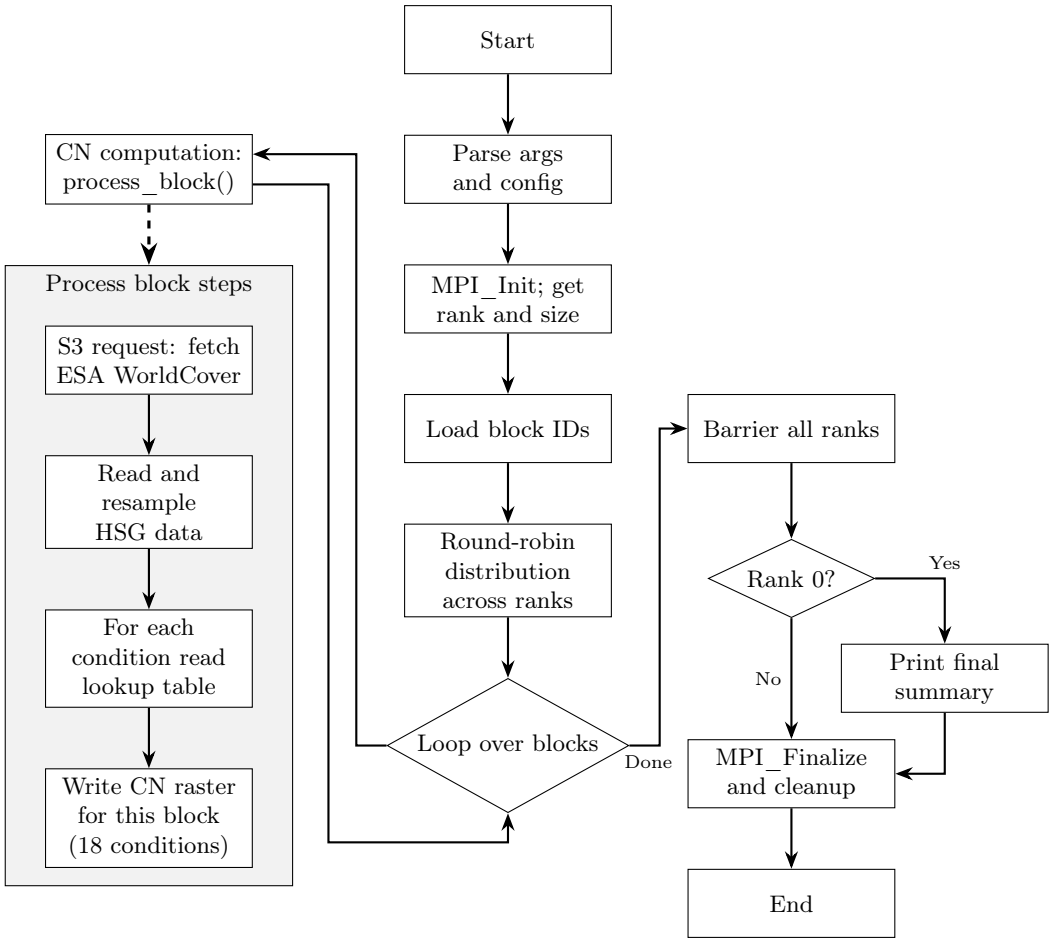


Figure 4: Workflow from configuration and MPI initialization through task dispatch, per-block kernel execution, and tiled output.

Table 5: Primary command-line flags.

Flag	Meaning
`-c <file>`	Path to configuration file
`-l <file>`	Text file with one ID per line
`-o`	Overwrite existing outputs (default is skip existing outputs)
`-v`	Verbose logging
`--help`	Show usage and exit
`--version`	Print version and exit

3. Illustrative examples

This example shows a simple end-to-end watershed workflow using the finalized MPI architecture. The study area is an example watershed in New Mexico. We

1. select blocks whose `esa_extent_blocks.shp` geometries intersect the watershed AOI;
2. call the compiled MPI backend via the Python wrapper to generate all 18 CN variants for those blocks;
3. mosaic one CN variant (here Fair HC and ARC II with undrained soils); and
4. clip the CN raster and the ESA land-cover raster to the watershed boundary for side-by-side inspection in QGIS.

Listing 1 implements this workflow in a compact Python driver.

In this example, `mpirun` is responsible for launching multiple ranks; the Python wrapper only forwards arguments to the compiled `gcn10` binary. The pipeline generates all 18 CN variants for the intersecting blocks, after which any individual variant can be mosaicked and clipped for visualization or further analysis. For this watershed, the drained and undrained outputs are identical because the drainage input does not indicate mapped drained soils in the study area; this reflects the watershed characteristics rather than a software issue. Figure 5a shows the clipped Fair/II, undrained CN raster for the watershed boundary. Figure 5b shows the clipped ESA WorldCover 2021 raster for the same watershed as land-cover context. Viewed together, the panels provide a direct visual check of how dominant tree cover corresponds to the prevalence of lower CN values across most of the watershed, while higher CN values align with built-up corridors, open areas, and water-related features.

For users who prefer shell scripts, the same workflow can be orchestrated without Python as shown in Listing 2.

4. Impact

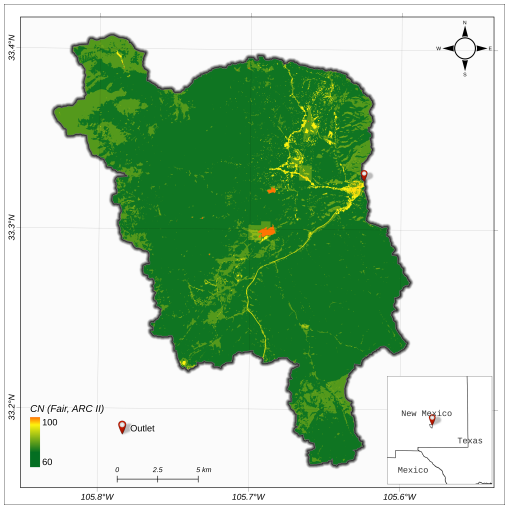
GCN10 turns high-resolution CN production from a weeks-long GIS exercise into a routine, hours-scale preprocessing step. On a single node, it delivers 10 m CN maps for large watersheds and multi-state studies, generates all 18 HC/ARC and drainage variants in one pass, and hands modelers ready-to-use rasters rather than a pile of intermediate tiles. Outputs are COGs with a fixed geometry and stable NoData, which makes mosaicking and clipping straightforward and repeatable across toolchains. Because the

```python
from pathlib import Path
from osgeo import ogr, osr
from subprocess import run
from gcn10py import run as gcn

cfg = Path("config.txt")
blocks = Path("esa_extent_blocks.shp")
aoi = Path("watershed.gpkg")
layer = "watershed"

def block_ids(blocks_path, aoi_path, layer_name, id_field="ID"):
    bds = ogr.Open(str(blocks_path)); bl = bds.GetLayer(0)
    ads = ogr.Open(str(aoi_path));    al = ads.GetLayerByName(layer_name)
    same = al.GetSpatialRef().IsSame(bl.GetSpatialRef())
    xform = None if same else osr.CoordinateTransformation(al.GetSpatialRef(), bl.GetSpatialRef())
    ids = set()
    for f in al:
        g = f.GetGeometryRef().Clone()
        if xform: g.Transform(xform)
        bl.SetSpatialFilter(g)
        for bf in bl:
            ids.add(bf.GetField(id_field))
        bl.SetSpatialFilter(None)
    return sorted(ids)

# 1) select blocks and run compiled backend under MPI (all 18 variants)
ids = block_ids(blocks, aoi, layer)
gcn(["-c", str(cfg), "-l", ",".join(map(str, ids)), "-o"])  # mpirun drives ranks externally

# 2) mosaic one CN variant
run(["gdalbuildvrt", "cn_fair_ii_ud.vrt", "./tiles/cn__fair_ii_*.tif"], check=True)

# 3) clip the CN raster to the watershed; preserve nodata=255
run([
    "gdalwarp", "-cutline", str(aoi), "-cl", layer, "-crop_to_cutline",
    "-dstnodata", "255", "cn_fair_ii_ud.vrt", "cn_fair_ii_ud_clip.tif"
], check=True)

# 4) clip the ESA land-cover raster for comparison in QGIS
run([
    "gdalwarp", "-cutline", str(aoi), "-cl", layer, "-crop_to_cutline",
    "-dstnodata", "0", "esa_worldcover_2021.vrt", "esa_worldcover_2021_clip.tif"
], check=True)
```

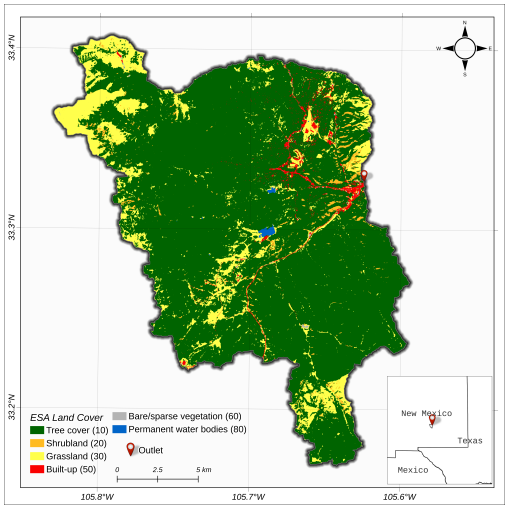
Listing 1: Minimal watershed driver: run CN generation with the compiled backend, mosaic one variant, and clip the CN and ESA rasters to the watershed boundary.

```
# select block IDs once (e.g., with GDAL/OGR), then:
mpirun -n 16 gcn10 -c config.txt -l blocks.txt -o

gdalbuildvrt cn_fair_ii_ud.vrt ./tiles/cn__fair_ii_*.tif

gdalwarp -cutline watershed.gpkg -cl watershed -crop_to_cutline \
         -dstnodata 255 cn_fair_ii_ud.vrt cn_fair_ii_ud_clip.tif

gdalwarp -cutline watershed.gpkg -cl watershed -crop_to_cutline \
         -dstnodata 0 esa_worldcover_2021.vrt esa_worldcover_2021_clip.tif
```

Listing 2: Equivalent shell-only workflow for generating the example CN raster and clipping the CN and ESA rasters to the watershed boundary.



(a) Fair/II, undrained CN raster clipped to the example watershed and displayed in QGIS. The panel shows the watershed boundary only, without subbasin aggregation; lower CN values dominate the forested interior, while higher values appear in built-up and more sparsely vegetated areas.



(b) ESA WorldCover 2021 land-cover raster clipped to the same watershed and displayed in QGIS. Tree cover is the dominant class, providing land-cover context for the lower CN values that dominate Figure 5a.

Figure 5: Example watershed visualization from the GCN10 workflow. The revised example emphasizes direct raster generation, mosaicking, and clipping. Together, the two panels show the CN pattern and the underlying land-cover context for the same watershed.

products are GIS-ready and can be clipped, inspected, and summarized into whatever hydrologic units a downstream workflow requires, hydrologic practitioners and model developers can shorten model-setup time, accelerate calibration and review cycles, and compare scenarios without rebuilding inputs.

The speed and determinism open workflows that were previously impractical. Practitioners can run ensembles over HCs and ARCs, and drainage, while still meeting project timelines, and they can probe sensitivity to seasonal land cover or alternative soils without sacrificing resolution. Quality control improves because file counts, names, and metadata follow a fixed pattern; resume-safe execution reduces risk on long runs and supports partial deliveries when deadlines loom. Cloud-optimized layout works well with modern GIS stacks and data lakes, enabling efficient downstream access, on-demand visualization, and scalable archive strategies.

The framework is open source and deliberately simple to adapt. Organizations can adjust lookup tables, add masks, or swap inputs without touching the parallel scaffolding, and the same rank-local pattern generalizes to other large-scale raster transforms where computation is light and I/O dominates. By lowering the cost and complexity of producing reproducible, high-resolution parameter maps at a continental scale, GCN10 expands access to consistent CN inputs for hydrologic modeling, especially for data-scarce regions where high-quality, up-to-date layers are most difficult to obtain.

5. Conclusions

GCN10 makes producing 10 m CN maps for large domains a routine, reproducible preprocessing task. The software combines a simple per-block CN kernel with a single-program, multiple-data MPI design, so each rank can process blocks independently without cross-rank communication. This choice keeps the implementation straightforward, makes outputs deterministic, and supports resume-safe production through stable filenames, a fixed NoData policy, and run manifests. Benchmarks show that the MPI-only implementation delivers large speedups on a single node, reducing the 32-block test from 7866 s with one rank to 845 s with 16 ranks. In contrast, hybrid MPI+OpenMP provides little additional benefit for this workload because end-to-end time is dominated by raster I/O and serialized write paths. In practice, GCN10 enables users to generate all 18 HC/ARC and drainage variants in one run, so they can move directly into watershed mosaicking, clipping, and user-defined summarization, including subbasin aggregation where needed for tools such as HEC-HMS and SWAT. The main remaining bottleneck is I/O throughput rather than computation. Future improvements will therefore focus on reducing read/write overhead (for example, more efficient batching and staging of input windows and faster compression choices for COGs creation) while preserving deterministic, restartable outputs. More broadly, the same block-parallel pattern can be reused for other large raster parameter pipelines and extended to support dynamic or seasonal CN products and alternative land-cover or soil inputs. Future application studies can also examine uncertainty propagation from source land-cover and HSG inputs, drainage assumptions, and lookup-table choices in specific hydrologic settings.

CRediT authorship contribution statement

Muhammad Abdullah Azzam: Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Software, Validation, Visualization, Writing—original draft, Writing—review & editing, **Huidae Cho**: Funding acquisition, Project administration, Resources, Supervision, Writing—review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was funded by the New Mexico Water Resources Research Institute (NM WRRI) and the New Mexico State Legislature under account number NMWRRI-SG-FALL2024. It utilized resources from the New Mexico State University High Performance Computing Group, which is directly supported by the National Science Foundation (OAC-2019000), the Student Technology Advisory Committee, and New Mexico State University and benefits from inclusion in various grants (DoD ARO-W911NF1810454; NSF EPSCoR OIA-1757207; Partnership for the Advancement of Cancer Research, supported in part by NCI grants U54 CA132383 (NMSU)).

References

[1] U.S. Department of Agriculture, Soil Conservation Service, Urban hydrology for small watersheds (TR-55), Technical Release 55, U.S. Department of Agriculture, Washington, DC, second edition (1986).
URL https://www.nrcs.usda.gov/sites/default/files/2019-07/TR55%20revised%201986.pdf

[2] U.S. Department of Agriculture, Natural Resources Conservation Service, National engineering handbook, part 630: Hydrology, chapter 9: Hydrologic soil-cover complexes, Tech. rep., U.S. Department of Agriculture, Washington, DC (2004).
URL https://directives.nrcs.usda.gov//sites/default/files2/1720460895/Chapter%209%20-%20Hydrologic%20Soil-Cover%20Complexes.pdf

[3] D. C. Garen, D. S. Moore, Curve number hydrology in water quality modeling: Uses, abuses, and future directions, Journal of the American Water Resources Association 41 (2) (2005) 377–388. doi:10.1111/j.1752-1688.2005.tb03742.x.
URL https://doi.org/10.1111/j.1752-1688.2005.tb03742.x

[4] C. W. Ross, L. Prihodko, N. P. Hanan, HYSOGs250m: Global gridded hydrologic soil groups for curve-number-based runoff modeling, Earth System Science Data 10 (2018) 1237–1244. doi:10.5194/essd-10-1237-2018.
URL https://doi.org/10.5194/essd-10-1237-2018

[5] Y. Wu, F. Kong, Q. Song, others, High-resolution annual dynamic dataset of Curve Number from 2008 to 2021 over conterminous united states, Scientific Data 11 (2024) 204. doi:10.1038/s41597-024-03044-2.

[6] J. G. Arnold, J. Kiniry, R. Srinivasan, M. J. White, J. R. Williams, D. K. Harmel, L. M. Haney, SWAT+ Theory and User Manual, Texas A&M AgriLife Research, Blackland Research & Extension Center (2018).
URL https://swat.tamu.edu

[7] X. Jin, Y. Jin, D. Yuan, X. Mao, Effects of land-use data resolution on hydrologic modelling, a case study in the upper reach of the Heihe river, northwest china, Ecological Modelling 404 (2019) 61–68. doi:10.1016/j.ecolmodel.2019.02.011.

[8] GRASS Development Team, Geographic resources analysis support system (GRASS GIS) software, version 8.4, Open Source Geospatial Foundation, USA (2024). doi:10.5281/zenodo.5176030.
URL https://grass.osgeo.org

[9] A. Azzam, r.curvenumber — GRASS GIS addons manual, Online manual, accessed 30 September 2025 (2025).
URL https://grass.osgeo.org/grass-stable/manuals/addons/r.curvenumber.html

[10] A. R. Siddiqui, SCS Curve Number (CN) Calculator, QGIS plugin, QGIS Plugin Repository (2020).
URL https://plugins.qgis.org/

[11] N. Gorelick, M. Hancher, M. Dixon, S. Ilyushchenko, D. Thau, R. Moore, Google earth engine: Planetary-scale geospatial analysis for everyone, Remote Sensing of Environment 202 (2017) 18–27. doi:10.1016/j.rse.2017.06.031.
URL https://doi.org/10.1016/j.rse.2017.06.031

[12] MPI Forum, Message passing interface: Version 4.1 (standard) (2024).
URL https://www.mpi-forum.org/mpi-41/

[13] D. Zanaga, R. Van De Kerchove, W. De Keersmaecker, others, ESA WorldCover 10 m 2020 v100, Zenodo (2021). doi:10.5281/zenodo.5571936.
URL https://doi.org/10.5281/zenodo.5571936

[14] U.S. Department of Agriculture, Natural Resources Conservation Service, National engineering handbook, part 630: Hydrology, chapter 10: Estimation of direct runoff from storm rainfall, Tech. rep., U.S. Department of Agriculture, Washington, DC (2004).
URL https://directives.nrcs.usda.gov/sites/default/files2/1720460920/Chapter%2010%20-%20Estimation%20of%20Direct%20Runoff%20from%20Storm%20Rainfall.pdf

[15] U.S. Army Corps of Engineers, HEC-HMS Hydrologic Modeling System, Version 4.x, User's Manual, Hydrologic Engineering Center, Davis, CA (2023).
URL https://www.hec.usace.army.mil/confluence/hms

[16] OpenMP Architecture Review Board, OpenMP application programming interface version 5.2 (2023).
URL https://www.openmp.org/specifications/

[17] G. M. Amdahl, Validity of the single-processor approach to achieving large-scale computing capabilities, in: AFIPS Conference Proceedings, Vol. 30, 1967, pp. 483–485. doi:10.1145/1465482.1465560.
URL https://doi.org/10.1145/1465482.1465560

[18] J. L. Gustafson, Reevaluating Amdahl's law, Communications of the ACM 31 (5) (1988) 532–533. doi:10.1145/42411.42415.
URL https://doi.org/10.1145/42411.42415

[19] GDAL/OGR Contributors, RFC 101: Raster dataset thread safety, GDAL Project Documentation (2024).
URL https://gdal.org/en/stable/development/rfc/rfc101_raster_dataset_threadsafety.html

[20] GDAL/OGR Contributors, VRT — GDAL virtual format (raster), GDAL Project Documentation (2024).
URL https://gdal.org/en/stable/drivers/raster/vrt.html

[21] Free Software Foundation, time(1) — linux man-pages 6.10, Online manual page, accessed 1 March 2025 (2024).
URL https://man7.org/linux/man-pages/man1/time.1.html

[22] S. Trecakov, N. Von Wolff, Doing more with less: Growth, improvements, and management of NMSU's computing capabilities, in: Practice and Experience in Advanced Research Computing (PEARC '21), ACM, Boston, MA, USA, 2021, p. 4 pages. doi:10.1145/3437359.3465610.
URL https://doi.org/10.1145/3437359.3465610

[23] GDAL/OGR Contributors, GDAL/OGR geospatial data abstraction library, version 3.x, Open Source Geospatial Foundation (2024).
URL https://gdal.org/

[24] Cloud Optimized GeoTIFF Project, Cloud optimized GeoTIFF: Overview and specification (2023).
URL https://www.cogeo.org

[25] N. Ritter, M. Ruth, The GeoTIFF data interchange standard, Version 1.0 (2000).
URL https://geotiff.org

[26] A. Azzam, gcn10py: Python interface to the high-performance C binary gcn10, GitHub repository (2025).
URL https://github.com/mabdazzam/gcn10py