File: **cp-r.py**

```
#!/usr/bin/env python3
"""
copy_rich_recursive.py: Recursively copy files or directories with a Windows-li
ke progress interface,
reporting totals in MiB, with smooth rendering and optional per-file bars only
for larger files.

Usage:
    copy_rich_recursive.py [OPTIONS] SRC [SRC ...] DEST

Options:
    --max-tasks N          Maximum number of active file bars to show (default:
 5)
    --min-display-size N   Minimum file size in MiB to show individual file bar
 (default: 1)

Examples:
    # Copy a folder, showing per-file bars only for files ≥1 MiB:
    copy_rich_recursive.py --max-tasks 5 --min-display-size 1 /src/dir /dest/di
r

Requirements:
    - pv (pipe viewer) in PATH
    - Python package: rich (`pip install rich`)

Behavior:
    - Displays a global total bar for all files.
    - Creates per-file bars only when file size ≥ min-display-size MiB.
    - Completed bars remain up to --max-tasks; smallest files skip per-file UI
to avoid flicker.
    - Prints a summary line: number of files and destination.
"""
import argparse
import json
import sys
from pathlib import Path
import subprocess

from rich.progress import (
    Progress, BarColumn, TextColumn, TimeElapsedColumn,
    TransferSpeedColumn, TimeRemainingColumn, ProgressColumn
)
from rich.text import Text

class MBColumn(ProgressColumn):
    """Displays completed/total in mebibytes (MiB)."""
    def render(self, task):
        comp = task.completed / (1024 * 1024)
        tot = task.total / (1024 * 1024) if task.total else 0
        return Text(f"{comp:.1f}/{tot:.1f} MiB")


def collect_files(sources):
    """
    Expand source paths to a flat list of files.
    Traverses directories recursively; files are included as-is.
    """
    all_files = []
    for src in sources:
        p = Path(src)
        if p.is_dir():
            for f in p.rglob('*'):
                if f.is_file():
                    all_files.append(f)
        elif p.is_file():
            all_files.append(p)
        else:
            print(f"Warning: {src} skipped", file=sys.stderr)
    return all_files


def build_dest_path(src_path, sources, dest_root):
    """
    Compute destination path preserving directory hierarchy.
    Top-level dirs recreate under dest_root.
    """
    for top in sources:
        top_p = Path(top)
        if top_p.is_dir() and Path(src_path).is_relative_to(top_p):
            rel = Path(src_path).relative_to(top_p)
            return dest_root / top_p.name / rel
    return dest_root / Path(src_path).name


def run_pv(src, dest_path, size, progress, global_task, file_task=None):
    """
```

```
    Invoke pv for one file, updating progress bars.
    If file_task is None, only global is updated.
    """
    last = 0
    pv_cmd = [
        "pv", "--numeric", "--wait",
        "--format", '{"bytes":%b}',
        "-s", str(size), str(src)
    ]
    with open(dest_path, 'wb') as out_f:
        proc = subprocess.Popen(
            pv_cmd, stdout=out_f, stderr=subprocess.PIPE, text=True
        )
        for line in proc.stderr:
            try:
                data = json.loads(line)
            except json.JSONDecodeError:
                continue
            done = data.get("bytes", 0)
            delta = done - last
            last = done
            if file_task is not None:
                progress.update(file_task, completed=done)
            progress.update(global_task, advance=delta)
        proc.wait()


def main():
    parser = argparse.ArgumentParser(
        description="Recursively copy with Rich UI, smoothing small-file flicke
r."
    )
    parser.add_argument('--max-tasks', type=int, default=5,
                        help='Max number of file bars to show')
    parser.add_argument('--min-display-size', type=float, default=1.0,
                        help='Minimum file size in MiB for per-file bar')
    parser.add_argument('sources', nargs='+',
                        help='Source files or dirs')
    parser.add_argument('dest', help='Destination directory')
    args = parser.parse_args()

    # Prepare destination
    dest_root = Path(args.dest)
    dest_root.mkdir(parents=True, exist_ok=True)

    # Collect files and get count
    files = collect_files(args.sources)
    if not files:
        print("No files found.", file=sys.stderr)
        sys.exit(1)
    file_count = len(files)
    print(f"Copying {file_count} files to {dest_root}")

    # Compute total bytes and min size threshold
    total_bytes = sum(f.stat().st_size for f in files)
    min_bytes = args.min_display_size * 1024 * 1024

    # Progress setup
    progress = Progress(
        TextColumn("[bold blue]{task.fields[filename]}", justify="right"),
        BarColumn(bar_width=None),
        TextColumn("{task.percentage:>3.0f}%"),
        TransferSpeedColumn(),
        TimeElapsedColumn(),
        TimeRemainingColumn(),
        TextColumn("[cyan]Total:"),
        BarColumn(bar_width=None),
        MBColumn(),
        refresh_per_second=10,
        expand=True
    )

    global_task = progress.add_task("global", filename="Total", total=total_byt
es)
    active_file_tasks = []

    with progress:
        for src in files:
            size = src.stat().st_size
            dest_path = build_dest_path(src, args.sources, dest_root)
            dest_path.parent.mkdir(parents=True, exist_ok=True)

            if size >= min_bytes:
                file_task = progress.add_task("file", filename=src.name, total=
size)
                active_file_tasks.append(file_task)
            else:
                file_task = None

            run_pv(src, dest_path, size, progress, global_task, file_task)

            # Cleanup
            if file_task is not None and len(active_file_tasks) > args.max_task
```

```
s:
                old = active_file_tasks.pop(0)
                progress.remove_task(old)

    print("Copy complete.")

if __name__ == '__main__':
    main()
```
