Recent Additions to HDF5

August 16, 2023

Neil Fortner, The HDF Group

Multi Dataset I/O

Multi Dataset I/O

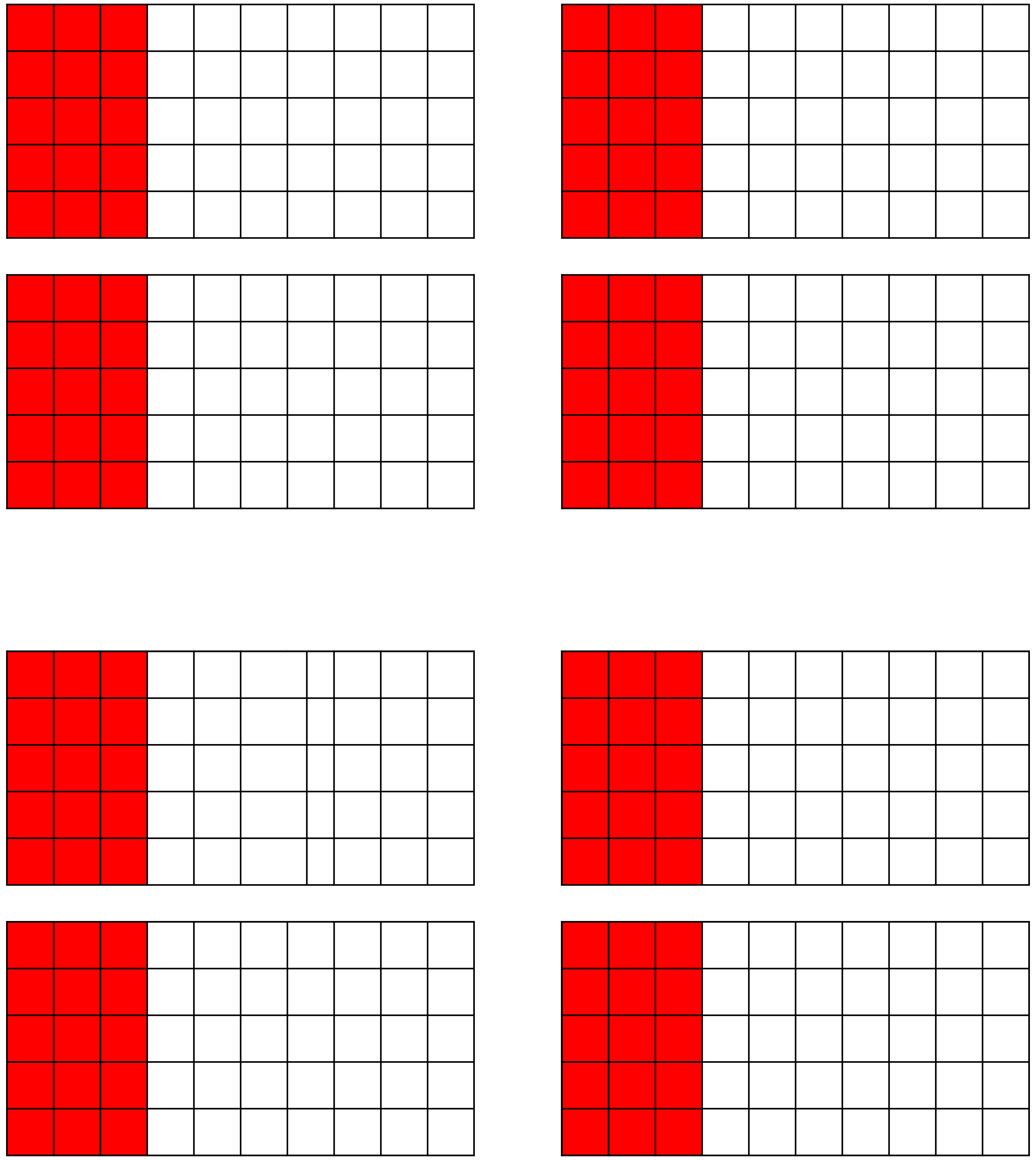
- Many applications perform I/O on multiple datasets
- Legacy API requires app to issue these I/O calls one dataset at a time
- New APIs available since 1.14.0 allow I/O to multiplie datasets in a single API call
- HDF5 library can, in many cases, aggregate all this information and pass it to the virtual file driver in a single callback
- Potential performance improvement

Example: MPI I/O

- Perform I/O on two datasets, each with 4 chunks
- Select the first 3 columns of each chunk

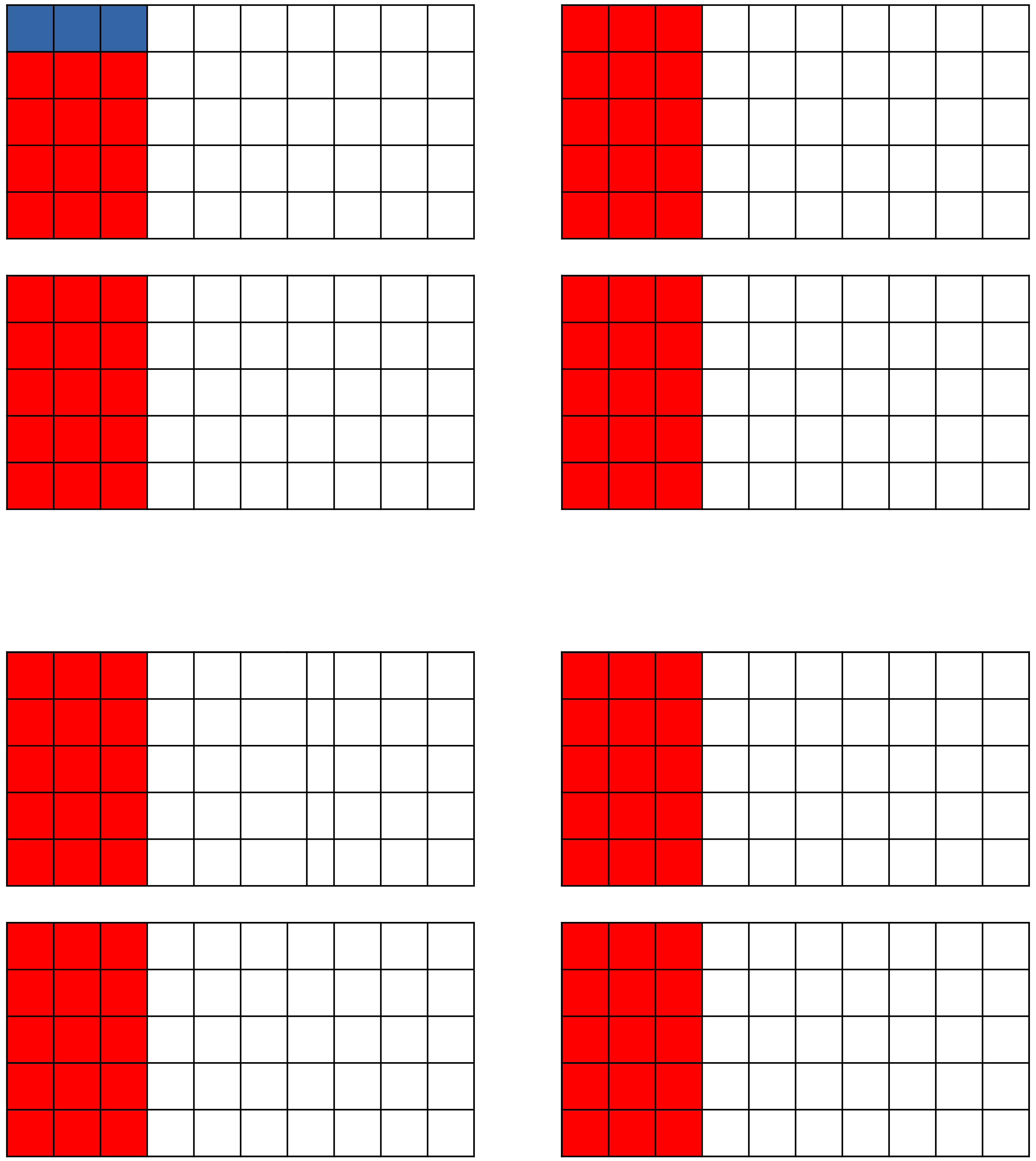
Independent I/O Example

- **Chunked dataset with partial I/O (red squares):**
  - One MPI_File_read/write_at() call per row, so **40** calls total



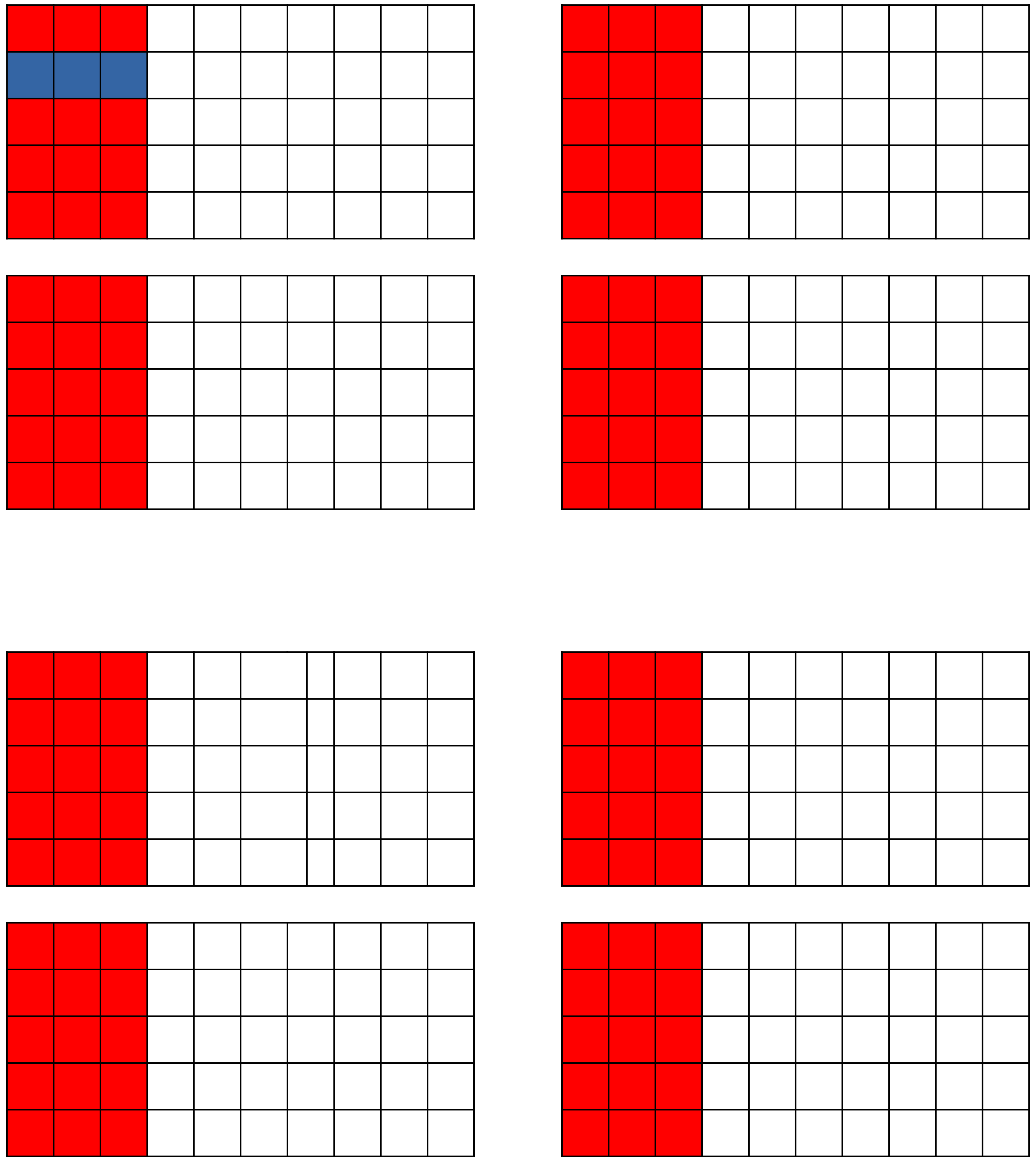
Independent I/O Example

- Chunked dataset with partial I/O (red squares):
  - One MPI_File_read/write_at() call per row, so **40** calls total



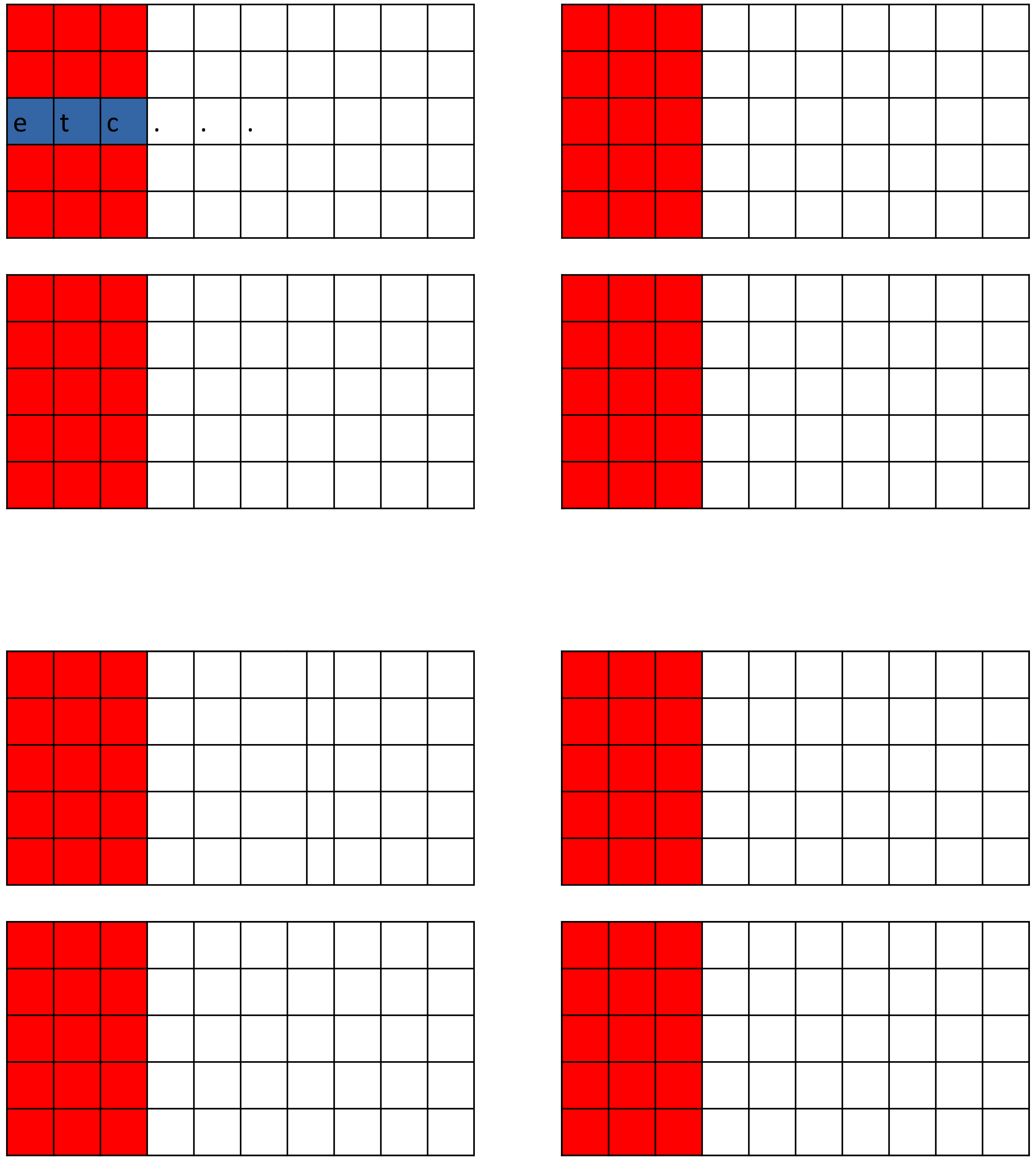
Independent I/O Example

- **Chunked dataset with partial I/O (red squares):**
  - One MPI_File_read/write_at() call per row, so **40** calls total



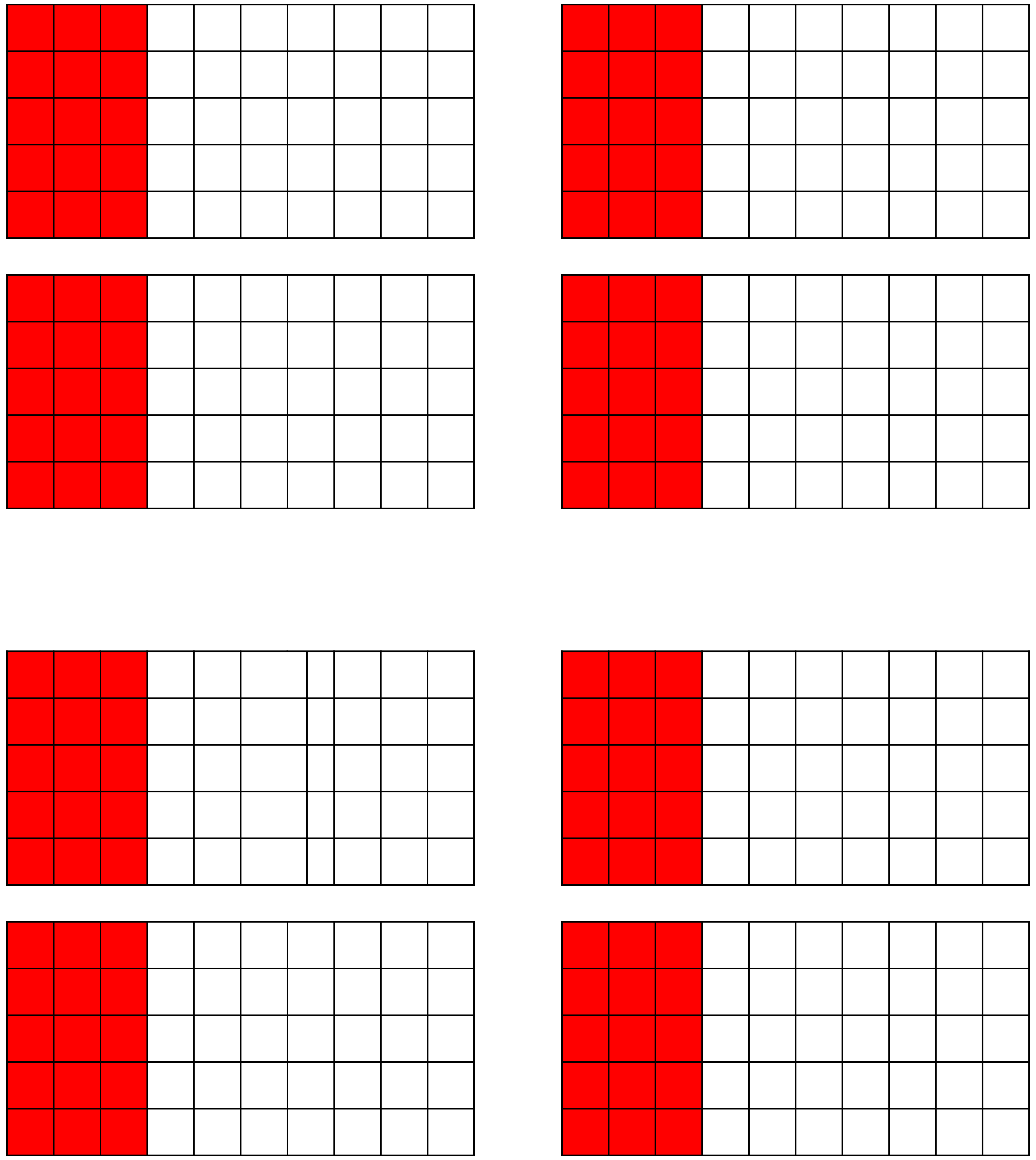
Independent I/O Example

- **Chunked dataset with partial I/O (red squares):**
  - One MPI_File_read/write_at() call per row, so **40** calls total



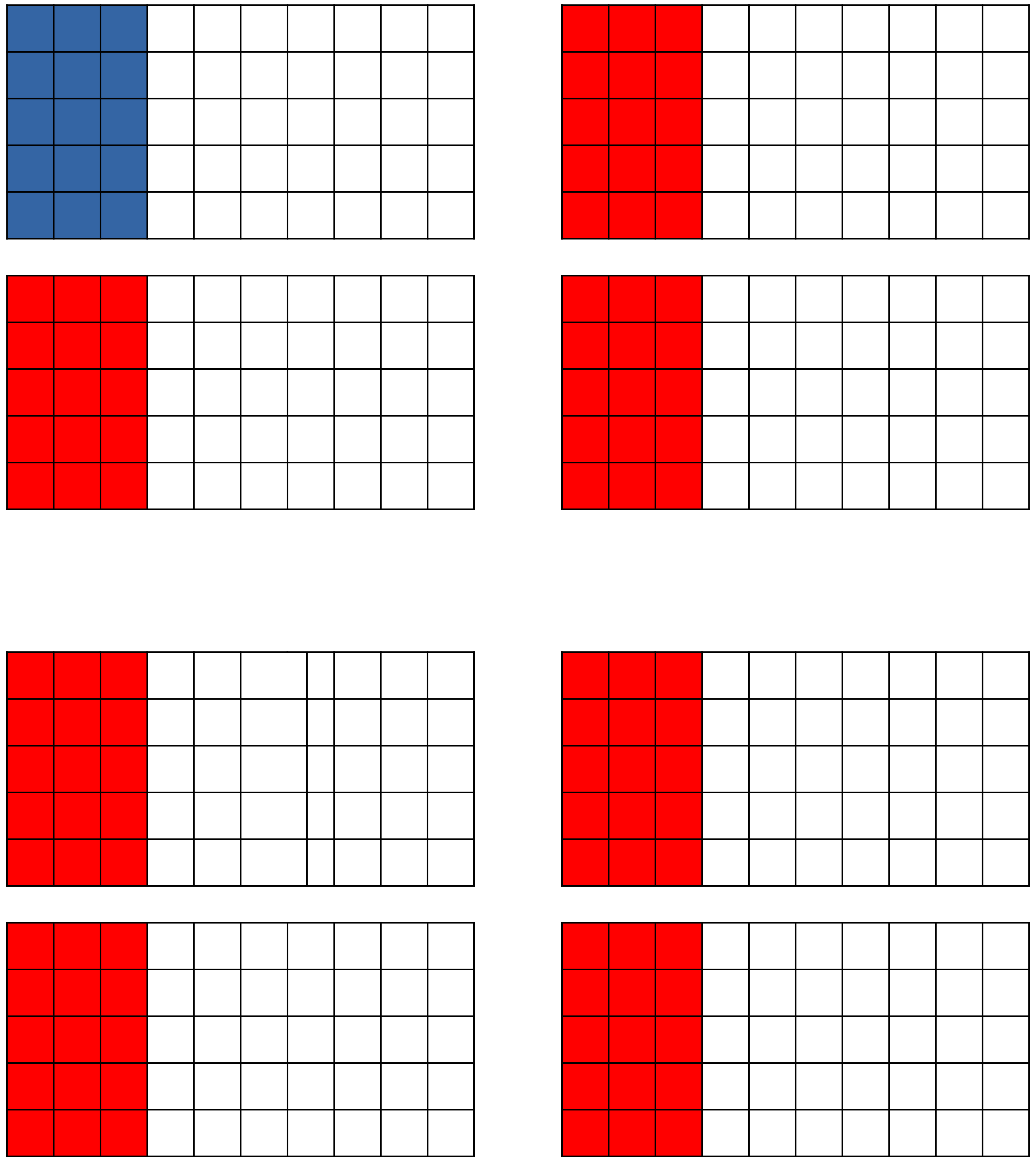
Multi-Chunk I/O Example

- **Chunked dataset with partial I/O (red squares):**
  - One MPI_File_read/write_at(_all)() call per chunk, so **8** calls total



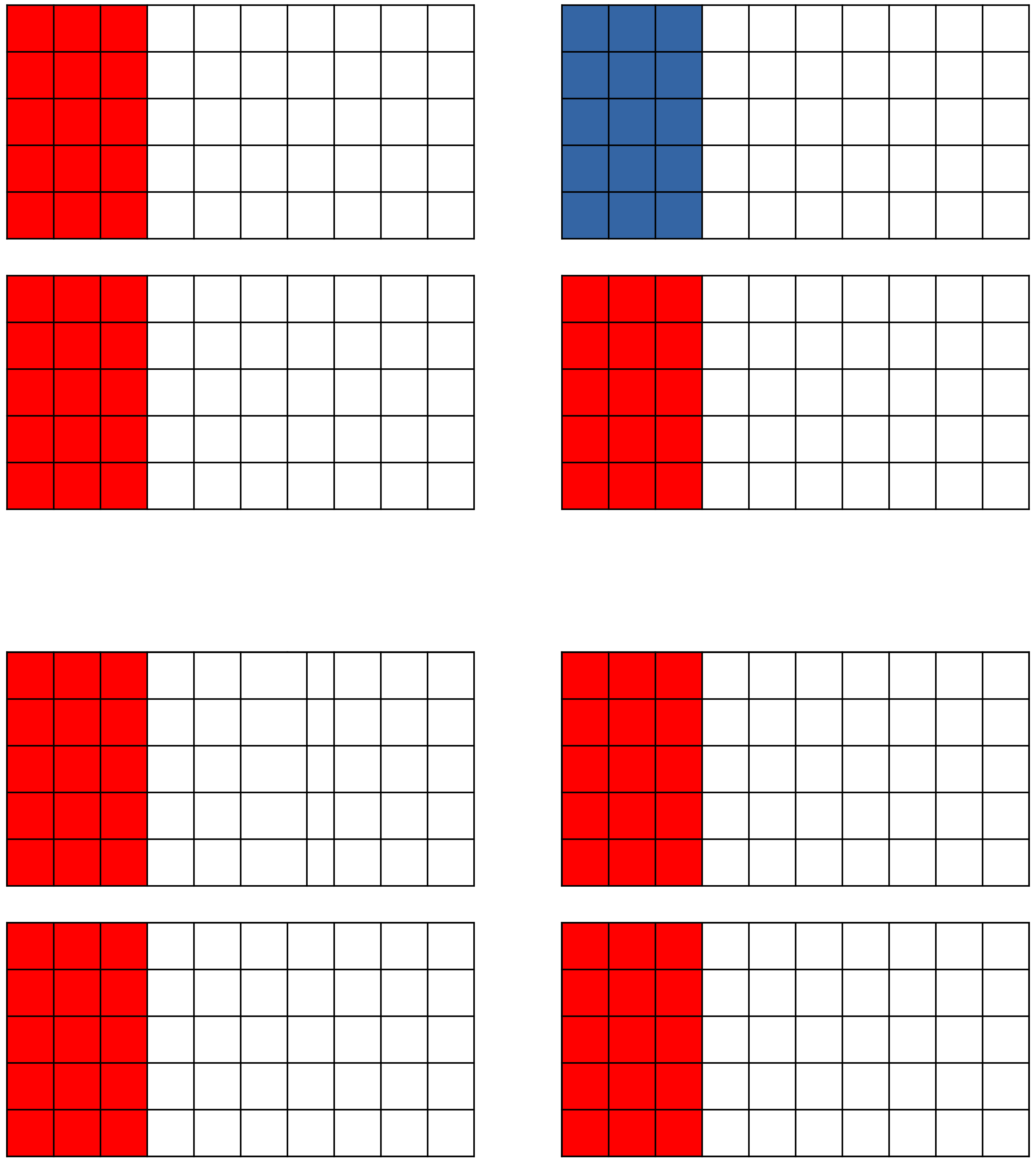
Multi-Chunk I/O Example

- **Chunked dataset with partial I/O (red squares):**
  - One MPI_File_read/write_at(_all)() call per chunk, so **8** calls total



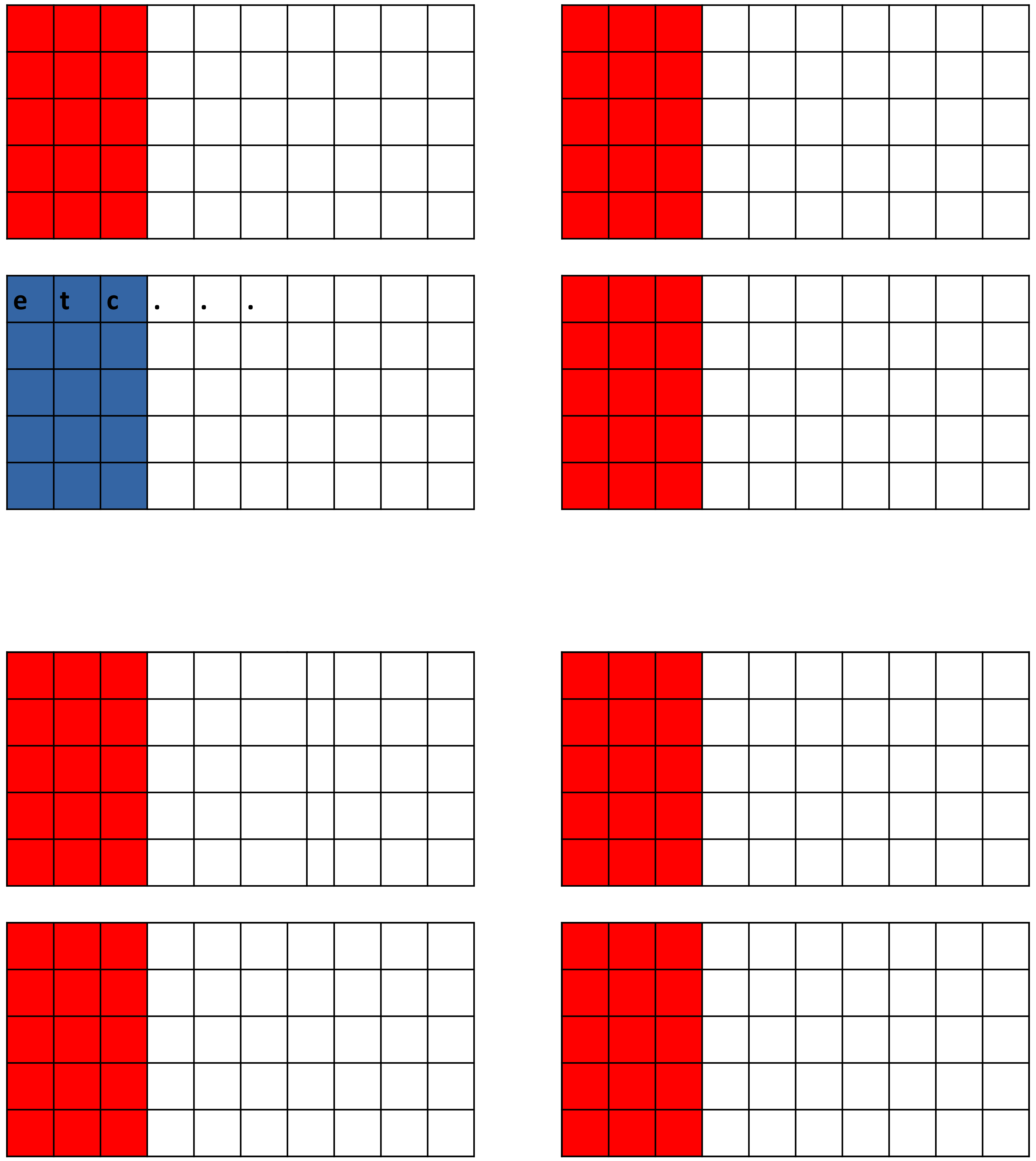
Multi-Chunk I/O Example

- **Chunked dataset with partial I/O (red squares):**
  - One MPI_File_read/write_at(_all)() call per chunk, so **8** calls total



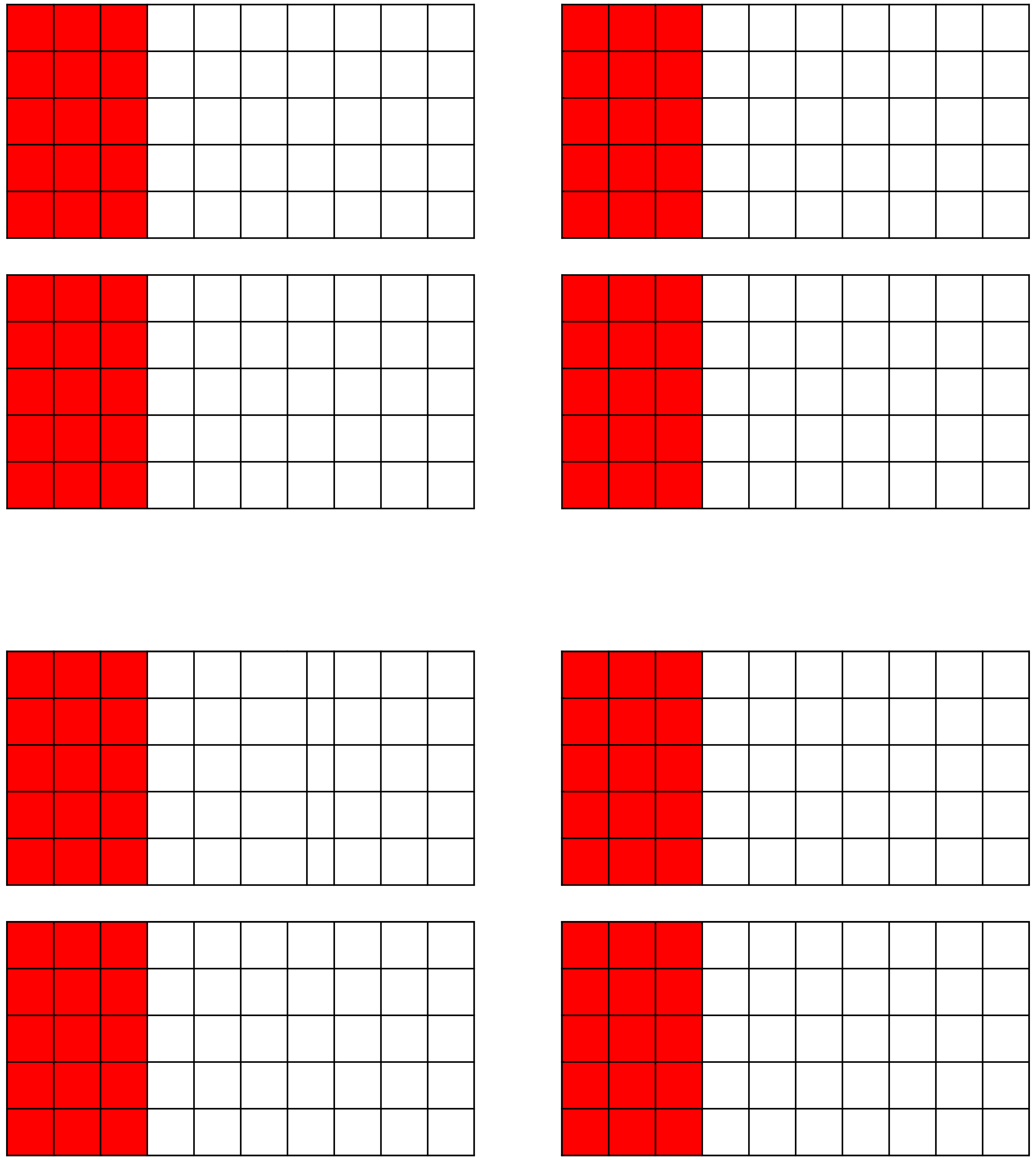
Multi-Chunk I/O Example

- Chunked dataset with partial I/O (red squares):
  - One MPI_File_read/write_at(_all)() call per chunk, so **8** calls total



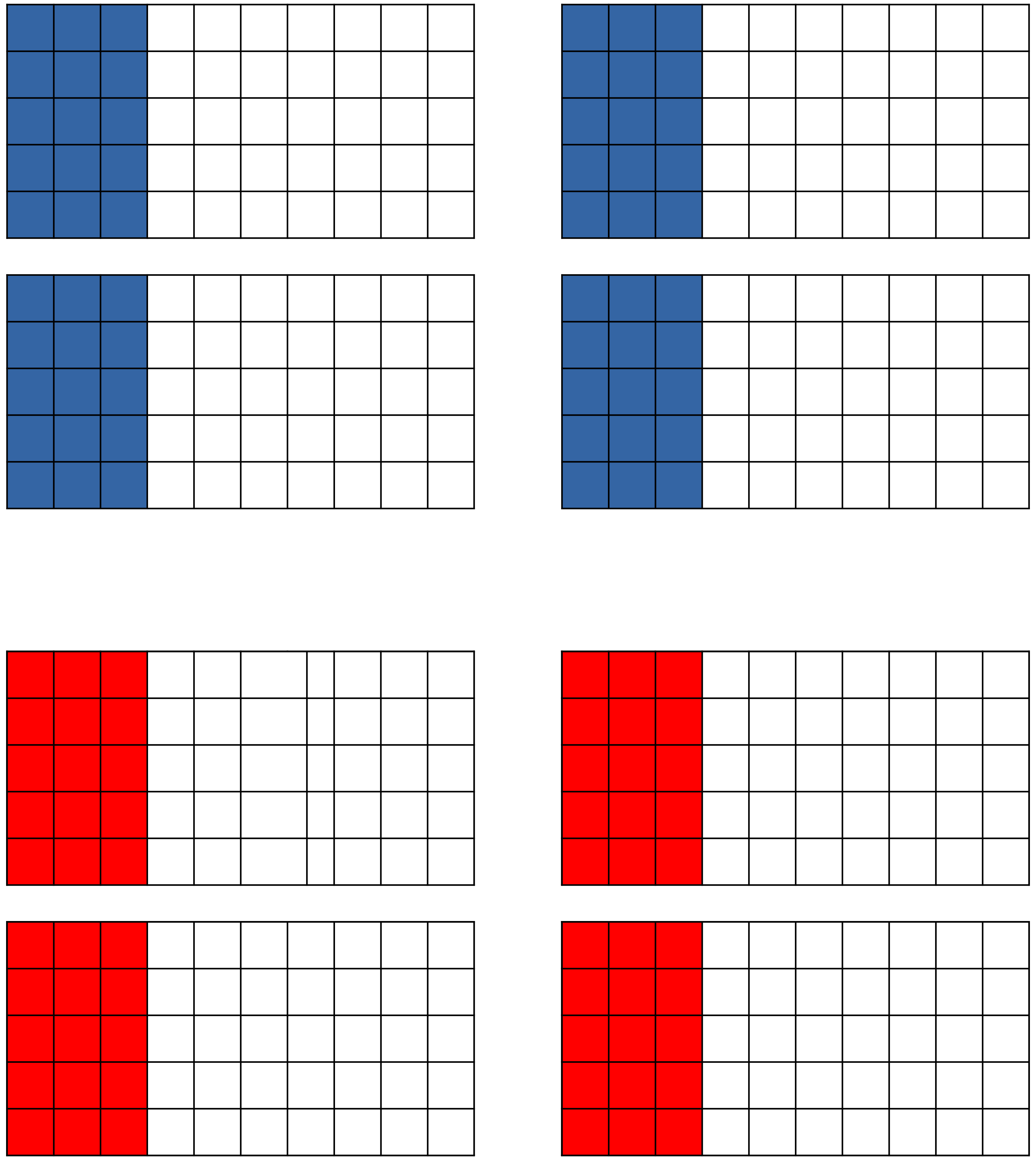
Link-Chunk I/O Example

- **Chunked dataset with partial I/O (red squares):**
    - One MPI_File_read/write_at(_all)() call per dataset, so **2** calls total



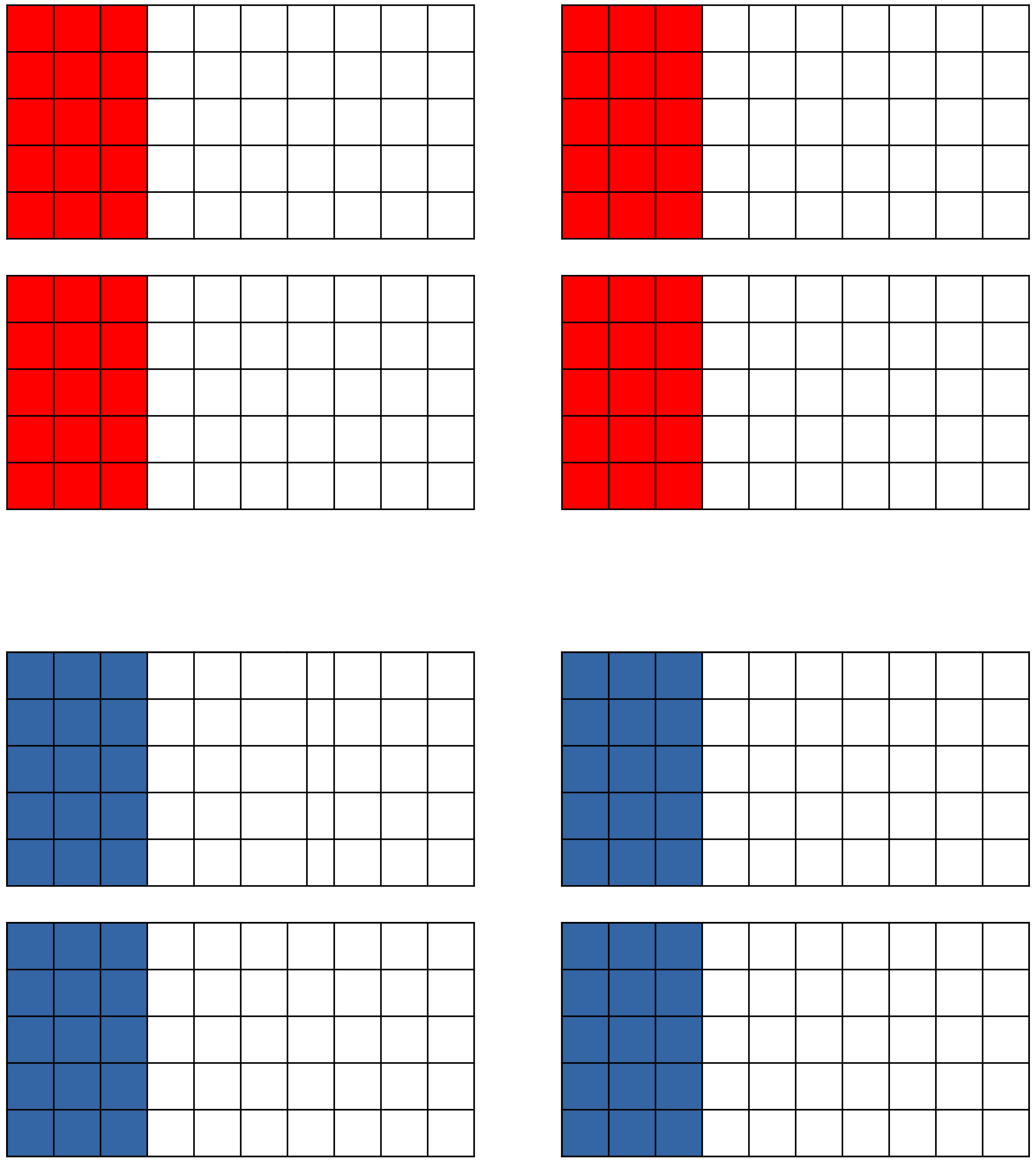
Link-Chunk I/O Example

- Chunked dataset with partial I/O (red squares):
  - One MPI_File_read/write_at(_all)() call per dataset, so **2** calls total



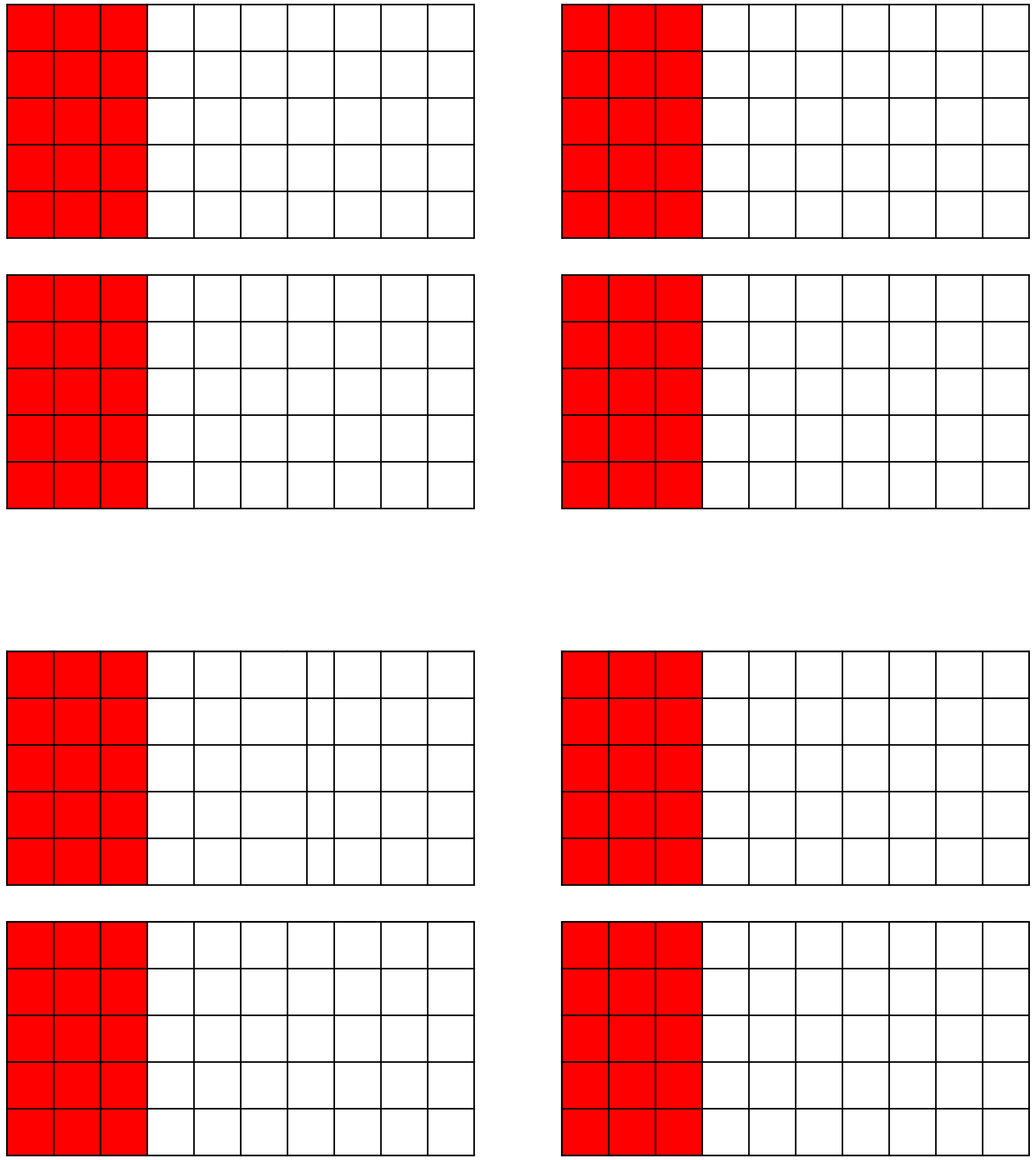
Link-Chunk I/O Example

- Chunked dataset with partial I/O (red squares):
  - One MPI_File_read/write_at(_all)() call per dataset, so **2** calls total



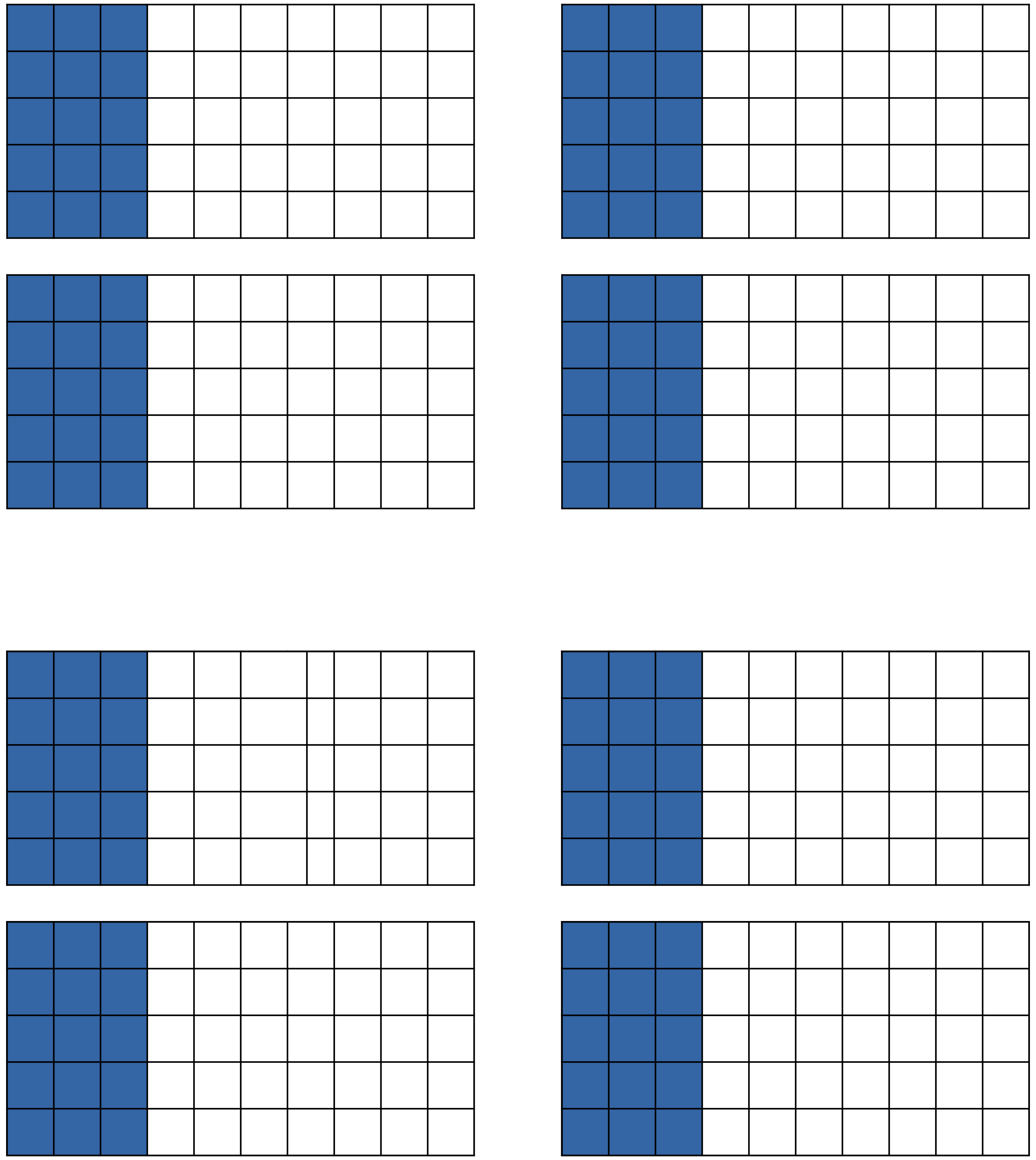
Multi Dataset I/O Example

- **Chunked dataset with partial I/O (red squares):**
  - One MPI_File_read/write_at(_all)() call per I/O, so **1** call total



Multi Dataset I/O Example

- **Chunked dataset with partial I/O (red squares):**
  - One MPI_File_read/write_at(_all)() call per I/O, so **1** call total



Multi Dataset I/O

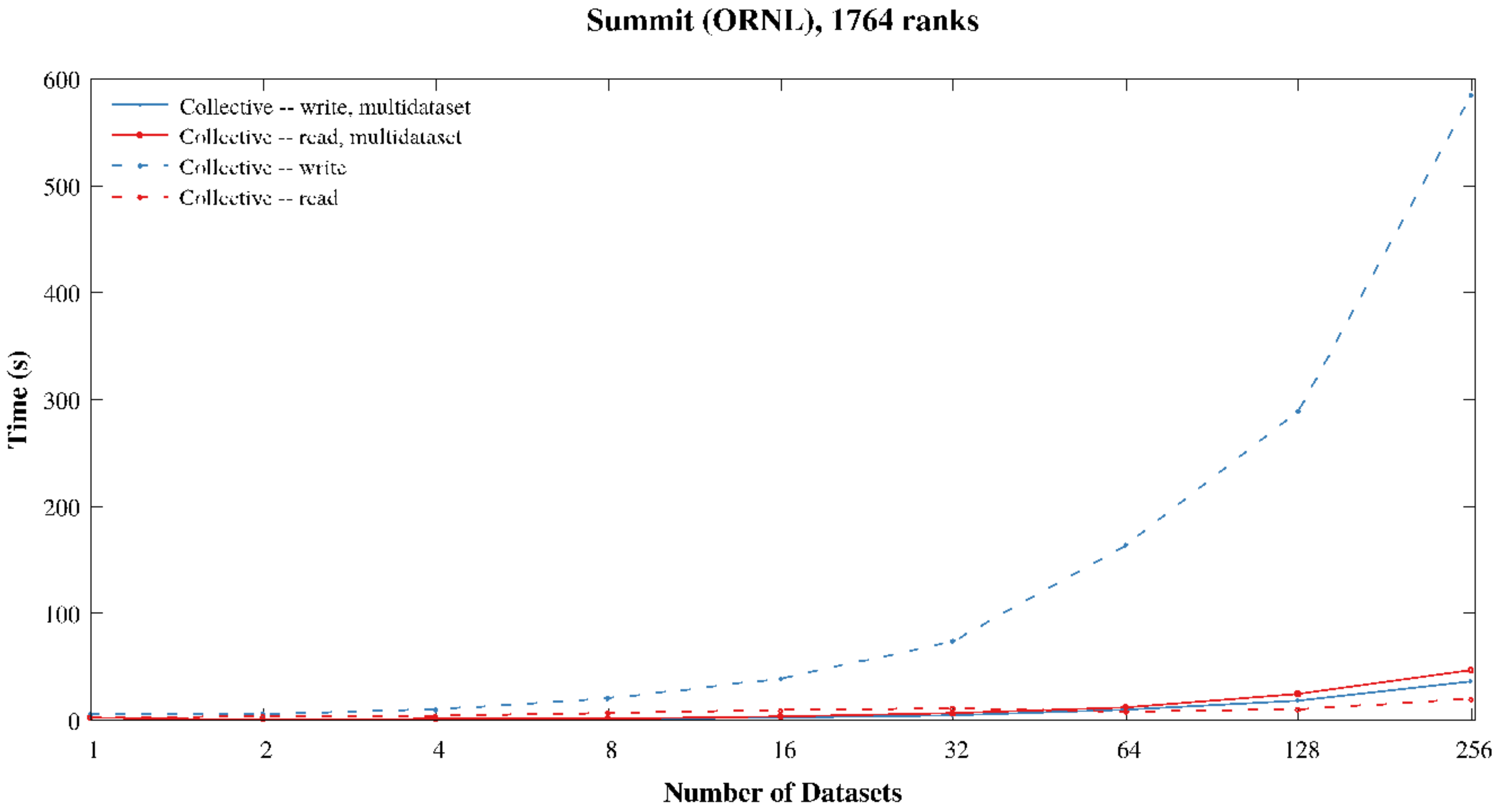
- API:

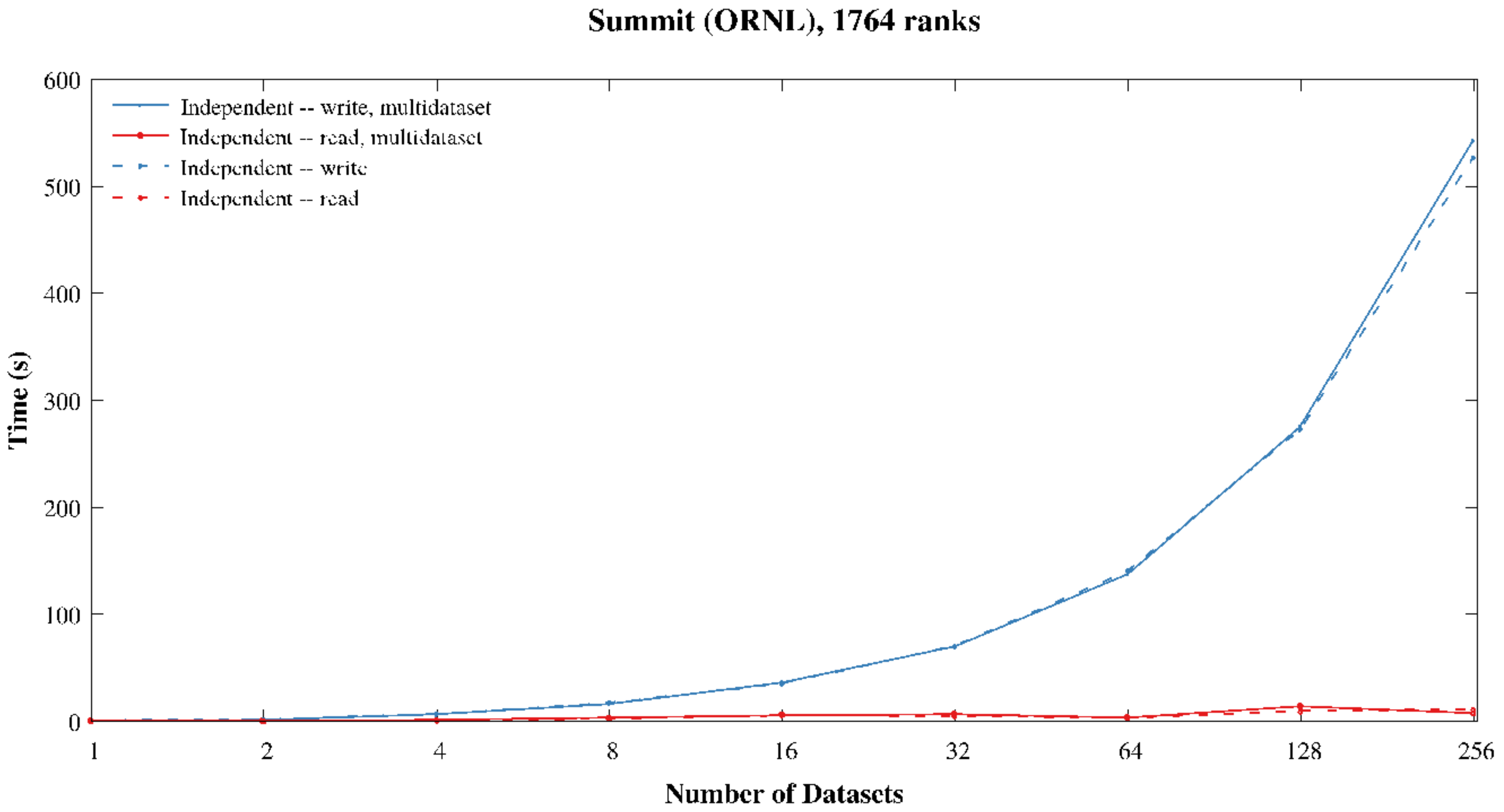
  -
    ```
    herr_t H5Dread_multi( hsize_t count, hid_t dataset_id[],
    hid_t mem_type_id[], hid_t mem_space_id[], hid_t
    file_space_id[], hid_t xfer_plist_id, void * buf[] )
    ```

  -
    ```
    herr_t H5Dwrite_multi( hsize_t count, hid_t dataset_id[],
    hid_t mem_type_id[], hid_t mem_space_id[], hid_t
    file_space_id[], hid_t xfer_plist_id, const void * buf[] )
    ```

Benchmark Results

- **Standalone Benchmark**
  - Constant number of ranks, vary number of datasets
  - Compare looped H5Dread/write with H5Dread/write_multi
  - 7 GiB per dataset

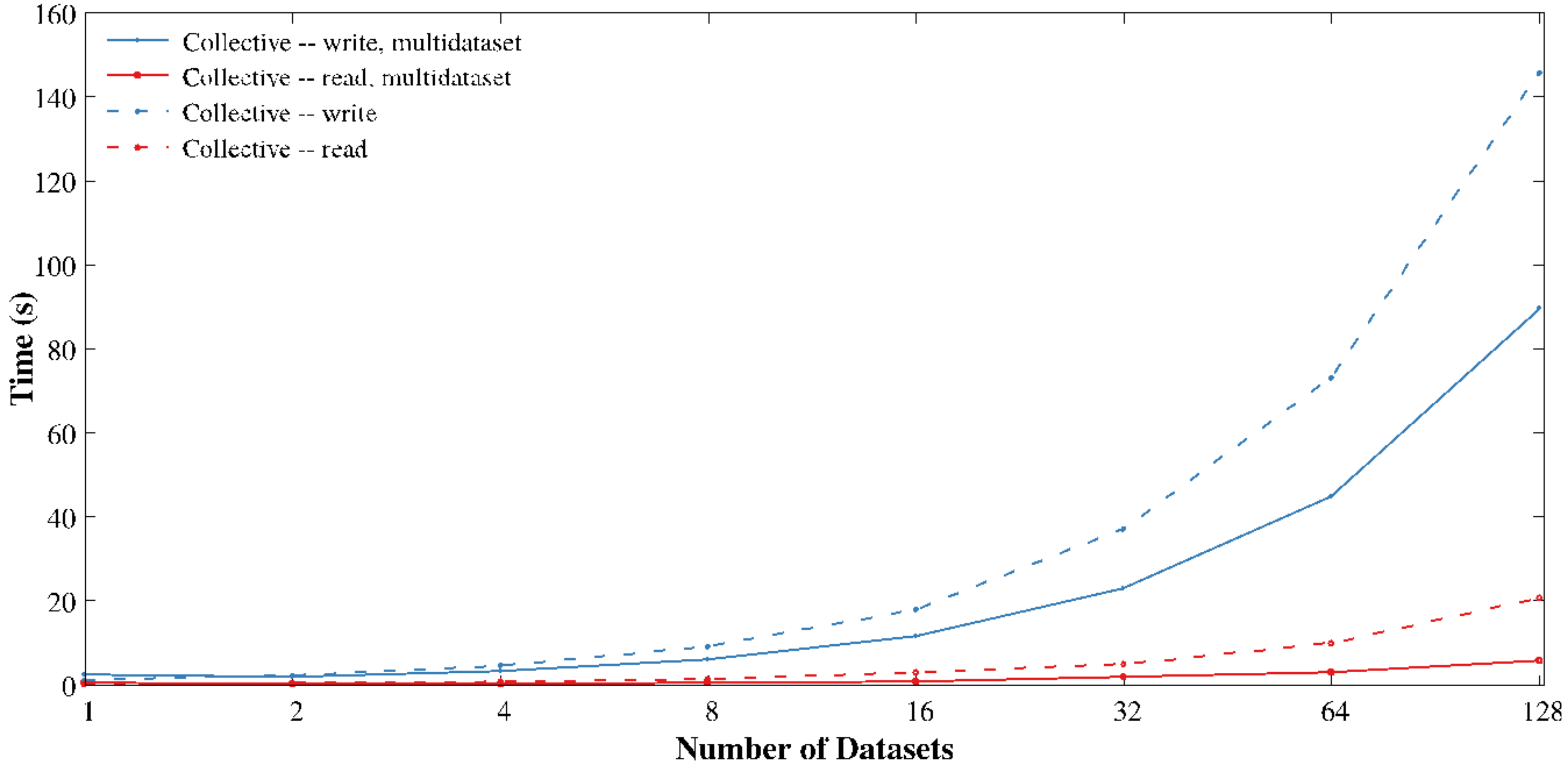




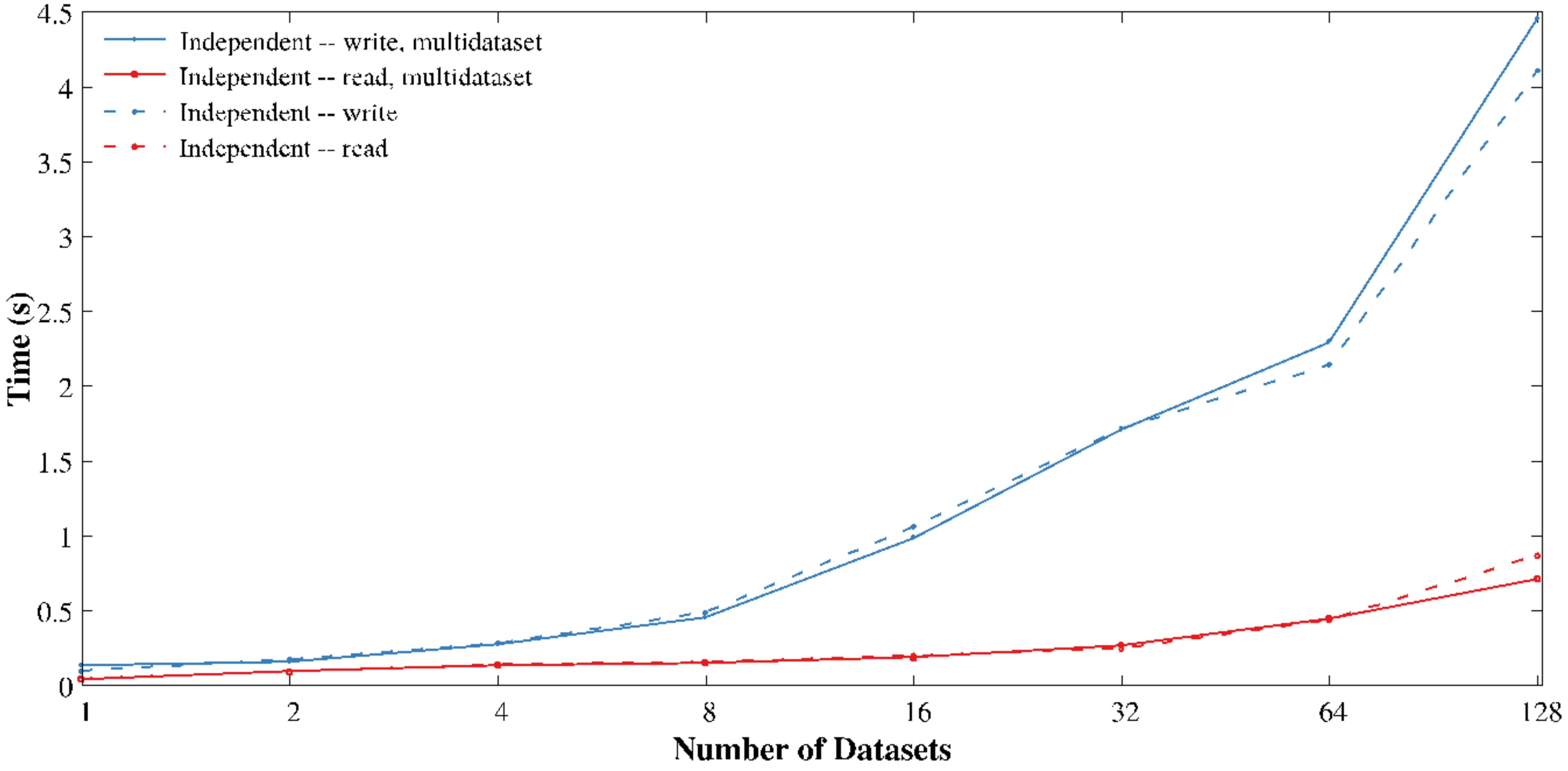
Benchmark Results

- **Standalone Benchmark**
  - Constant number of ranks, vary number of datasets
  - Compare looped H5Dread/write with H5Dread/write_multi
  - 7 GiB per dataset

Polaris (ANL), 2048 ranks



Polaris (ANL), 2048 ranks



Benchmark Results

- **Quick CGNS Benchmark**
  - 16 H5Dread/write() calls -> 6 H5Dread/write_multi() calls (don’t expect huge improvement)
  - On Summit, with problem size held constant:
    - 2688 ranks, ~10% improvement
    - 10752 ranks, ~6% improvement

Supported Use Cases

- All ranks must pass the same list of datasets (in collective mode)
- All datasets must be in the same file
- Each dataset may only be present once in the list
- Selection I/O fully supported
- For simultaneous multi dataset I/O:
  - Must be in collective mode – H5Pset_dxpl_mpio
  - None of the datasets can have data filters/compression (ongoing work!)
  - All datasets must have contiguous or chunked layout
  - Otherwise, library will process one dataset at a time

Questions?

- Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Selection and Vector I/O

Selection and Vector I/O

- Before 1.14.1, all virtual file drivers (VFDs) except the built in MPIO driver would only accept a single offset/length pair per operation
  - Effectively like the “independent” example shown for multi dataset
  - For the MPIO VFD, the library detects if it is in use, and if so, constructs MPI datatypes for the I/O and passes it to the VFD through undocumented DXPL properties
- We wanted to allow other VFDs, including external VFDs, to handle non-contiguous I/O in an intelligent fashion
- We have added selection and vector I/O callbacks to the VFD struct in 1.14.1

Vector I/O

- Vector I/O refers to passing a list of offsets and lengths to the VFD layer instead of a single pair
- Simple, but potentially costly in terms of memory usage, even if the offsets/lengths are regularly spaced
-

```
    herr_t (*read_vector)(H5FD_t *file, hid_t dxpl, uint32_t count, H5FD_mem_t
types[], haddr_t addrs[], size_t sizes[], void *bufs[]);
    herr_t (*write_vector)(H5FD_t *file, hid_t dxpl, uint32_t count, H5FD_mem_t
types[], haddr_t addrs[], size_t sizes[], const void *bufs[]);
```

Selection I/O

- In selection I/O we instead pass a list of offsets and HDF5 dataspace selections
- Allows more compact representation of regular selections
- For non-regular selections, query currently effectively reduces to offset/length pairs (new API being considered to improve this)
-
```
    herr_t (*read_selection)(H5FD_t *file, H5FD_mem_t type, hid_t dxpl_id, size_t
count, hid_t mem_spaces[], hid_t file_spaces[], haddr_t offsets[], size_t
element_sizes[], void *bufs[] /*out*/);

    herr_t (*write_selection)(H5FD_t *file, H5FD_mem_t type, hid_t dxpl_id, size_t
count, hid_t mem_spaces[], hid_t file_spaces[], haddr_t offsets[], size_t
element_sizes[], const void *bufs[] /*in*/);
```

Supported Cases

- Supports multi dataset
- Support type conversion (see later slide)
- Conditions
  - Must be supported by VFD
    - MPIO VFD supports vector I/O since 1.14.1, selection I/O since 1.14.2
  - Must be chunked or contiguous dataset
  - Must not use data filters/compression (except for parallel I/O)
  - Must not use the chunk cache, sieve buffer, or page buffering
- Turn selection/vector I/O on and off with:

```
herr_t H5Pset_selection_io(hid_t plist_id, H5D_selection_io_mode_t
selection_io_mode);
```

Type Conversion

- Type conversion is supported with selection/vector I/O
  - This allows type conversion with parallel collective I/O, which was not possible before!
- However, the conversion buffer (and background buffer, if applicable) must be large enough to hold all elements involved in I/O, otherwise selection I/O is disabled
- To mitigate this, we have implemented in-place type conversion, where the application buffer is also used as the type conversion buffer, eliminating the need to allocate another large buffer.

In-Place Type Conversion

- For in-place type conversion to work, the memory type must not be smaller than the file type
- In addition, the memory selection corresponding to each chunk or contiguous dataset in the I/O must be contiguous
- For write operations, in-place type conversion will destroy data in the write buffer, so it is off by default. To allow in-place type conversion use:

```
herr_t H5Pset_modify_write_buf(hid_t plist_id, hbool_t modify_write_buf);
```

- In 1.14.2 this has been extended to also work with legacy (scalar) I/O
  - Allows a large I/O with type conversion in a single VFD call without needing a large type conversion buffer. Previously would always break into smaller operations.

Questions?

- Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Subfiling VFD

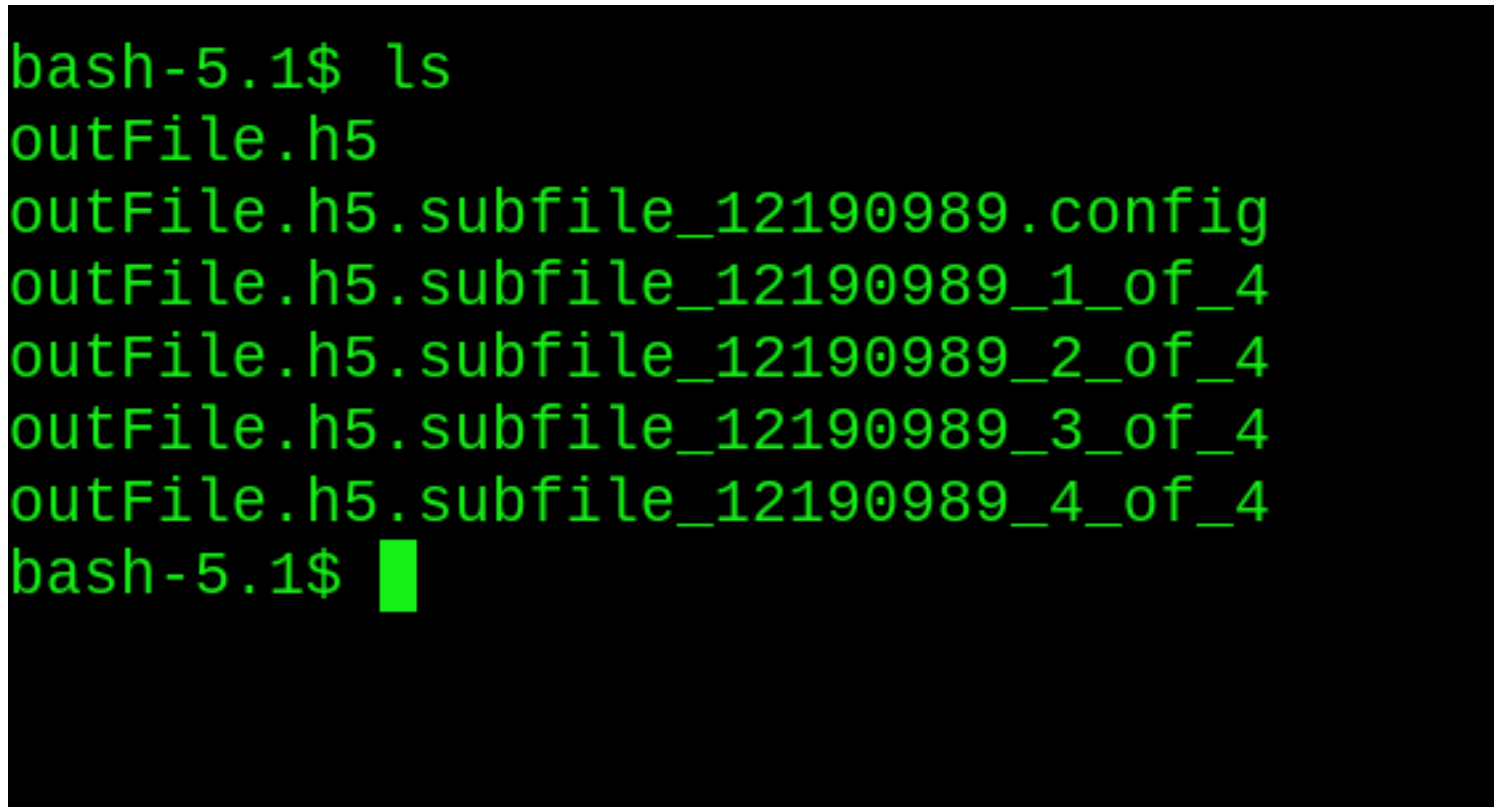
What is it?

- An MPI-based parallel file driver that allows an HDF5 application to distribute an HDF5 file across a collection of subfiles in equal-sized data segment stripes
  - Data stripe size is the amount of data (in bytes) that can be written to a subfile before data is placed in the next subfile in round-robin fashion
  - Defaults to 1 subfile per machine node with 32MiB data stripes
- Try to find a middle ground between single shared file and file-per-process approaches to parallel I/O
  - Minimize the locking issues of single shared file approach
  - Avoid some complexity and reduce total number of files compared to file-per-process approach
  - Designed to be flexible and configurable for different machines

What is it? (continued)

- Uses a system of "I/O concentrators" - subset of available MPI ranks which control subfiles and operate I/O worker thread pools
  - N-to-1 mapping from subfiles -> I/O concentrator ranks
  - Subfiles are assigned round-robin across the available I/O concentrator ranks, as determined by the chosen I/O concentrator selection method
  - I/O from non-I/O-concentrator MPI ranks is forwarded to the appropriate I/O concentrator based on offset in the logical HDF5 file

- Outputs several files per logical HDF5 file
  - HDF5 stub file
  - Subfiling VFD configuration file
  - Subfiles

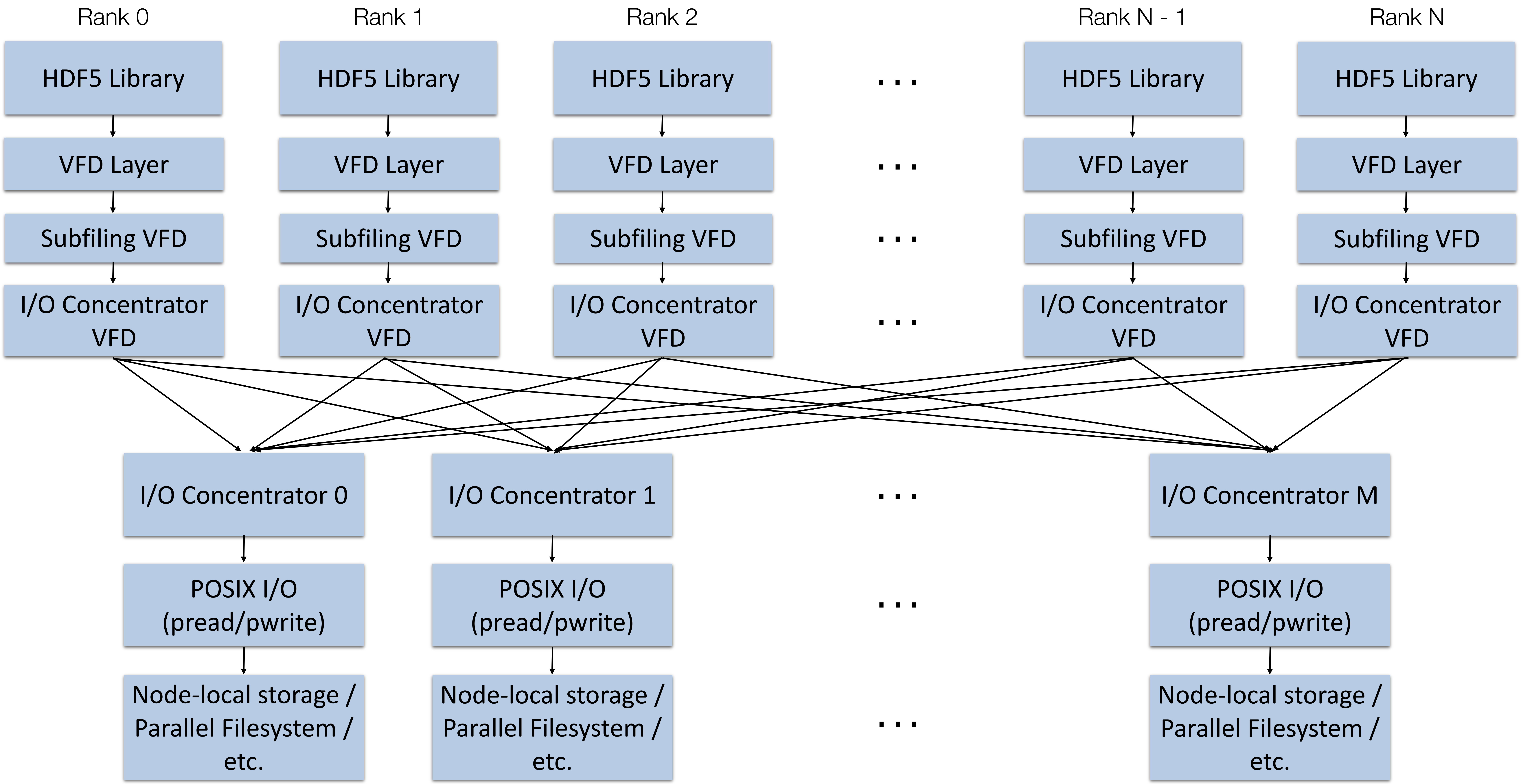


```
bash-5.1$ ls
outFile.h5
outFile.h5.subfile_12190989.config
outFile.h5.subfile_12190989_1_of_4
outFile.h5.subfile_12190989_2_of_4
outFile.h5.subfile_12190989_3_of_4
outFile.h5.subfile_12190989_4_of_4
bash-5.1$
```

Current Architecture

- Subfiling VFD stacked on top of I/O Concentrator VFD on each MPI rank
  - Subfiling VFD manages subfiling information (data stripe size, subfile count, HDF5 stub file, etc.) and breaks down I/O requests into offset/length vectors based on data stripe size and file offset
  - I/O Concentrator VFD receives I/O vectors and queues I/O calls to appropriate I/O concentrator
- Subset of MPI ranks selected as I/O concentrators
  - Each controls one or more subfiles
  - Receive I/O calls from I/O concentrator VFDs, translate to subfile-local file offsets and relay I/O call to appropriate subfile

Current Architecture



New API Calls

```
herr_t
H5Pset_fapl_subfiling(hid_t fapl_id, const H5FD_subfiling_config_t *vfd_config);
```

Modifies the given File Access Property List to use the Subfiling VFD and configures the VFD according to the parameters set in the specified subfiling configuration structure. The subfiling configuration structure may be NULL, in which case default values are used.

```
herr_t
H5Pget_fapl_subfiling(hid_t fapl_id, H5FD_subfiling_config_t *config_out);
```

Returns the subfiling parameters that were set on the given File Access Property List, or default values if no subfiling parameters were set

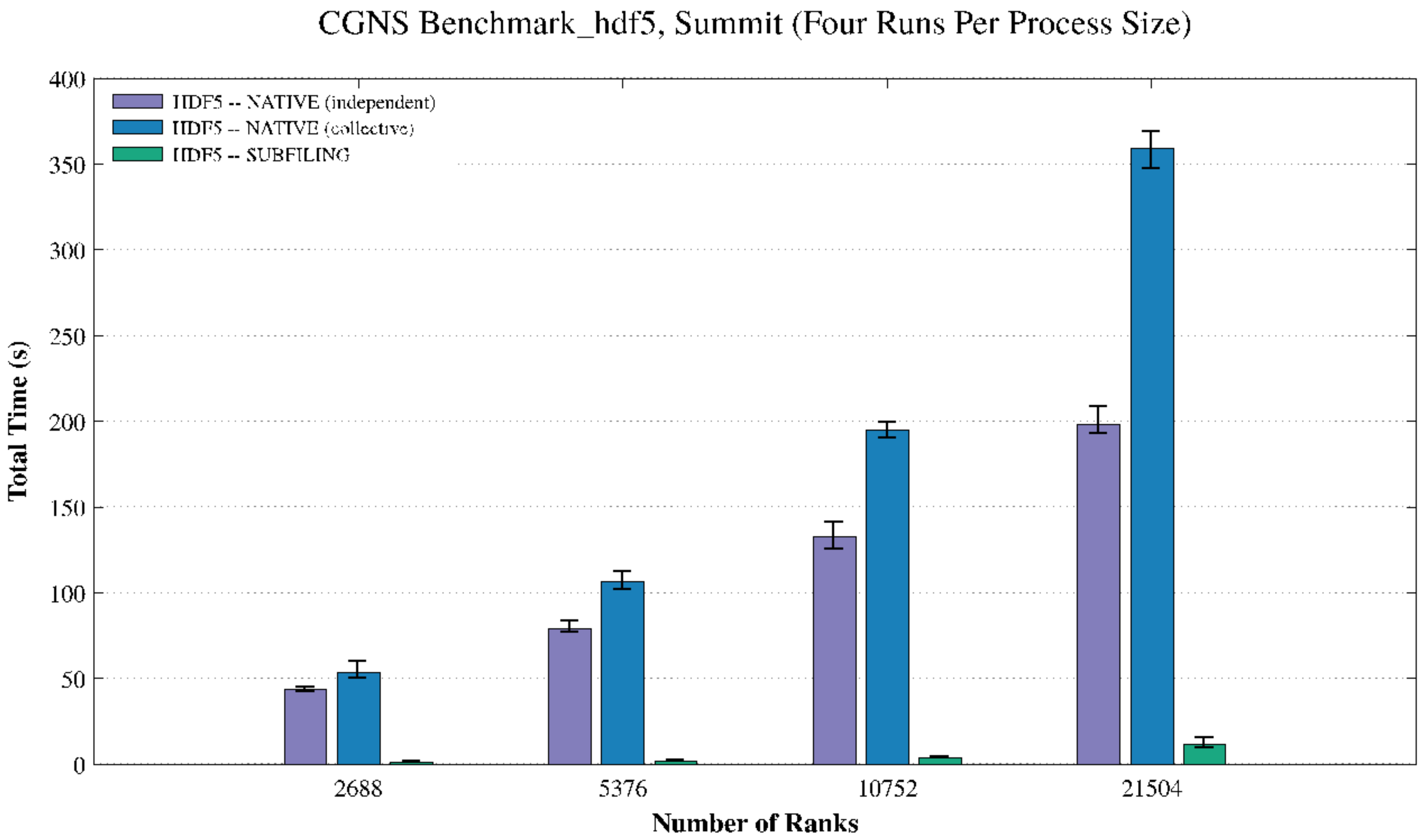
Performance Results

- CGNS = **C**omputational Fluid Dynamics (CFD) **G**eneral **N**otation **S**ystem
  - Standardize CFD I/O.
  - Subfiling version in the subfiling branch of the CGNS library.
  - *Benchmark_hdf5.c* writes and reads: mesh coordinates, element connectivity and solution data.
    - Summit (GPFS), using a mesh of 130 million (for 21k ranks), 6-node pentahedral elements.
      - The number of elements is halved as the ranks are decreased.

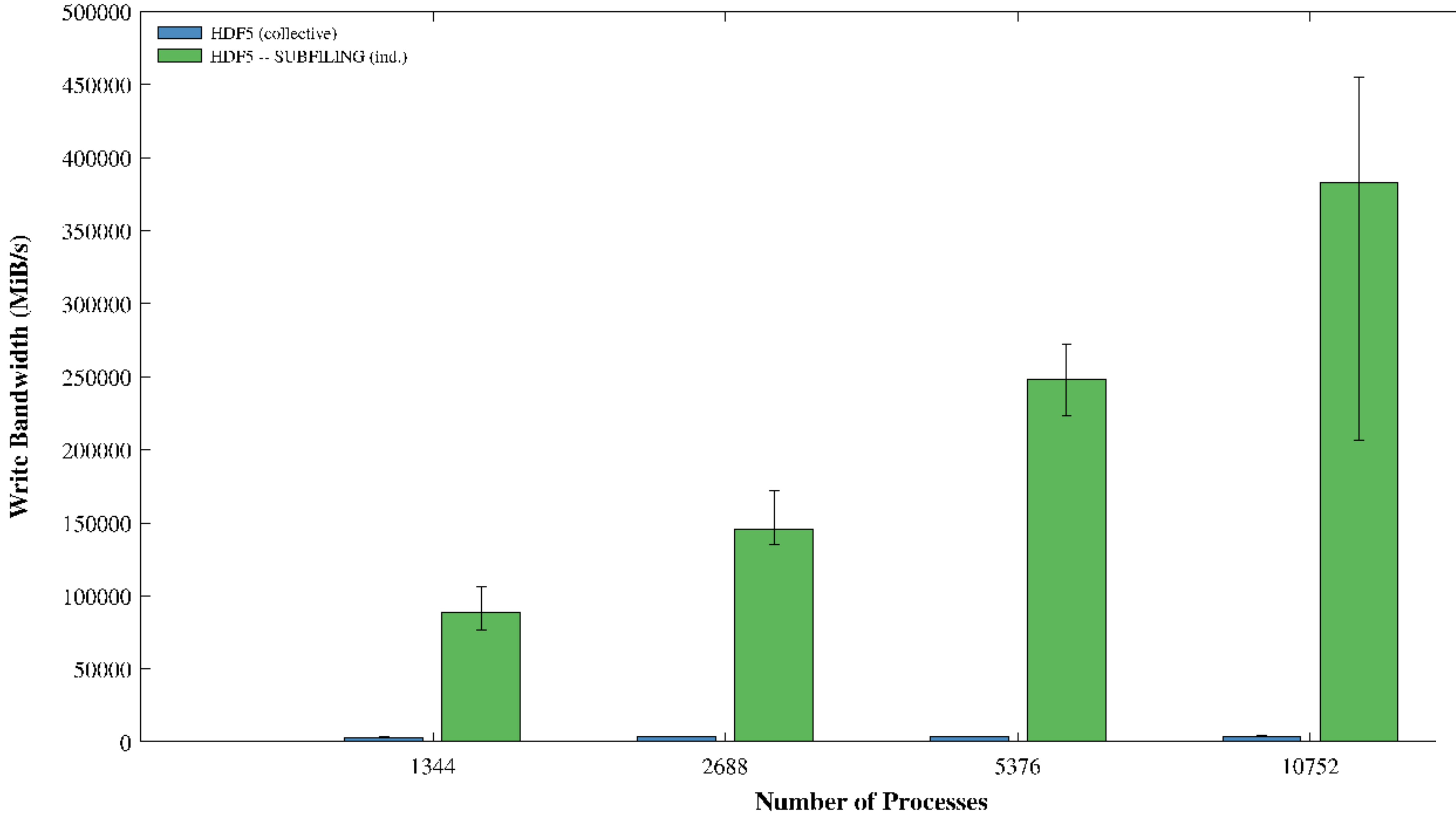
Number of Ranks	HDF5 File Size
21504	53 GiB
10752	27 GiB
5376	14 GiB
2688	6.6 GiB

Performance Results

The HDF Group

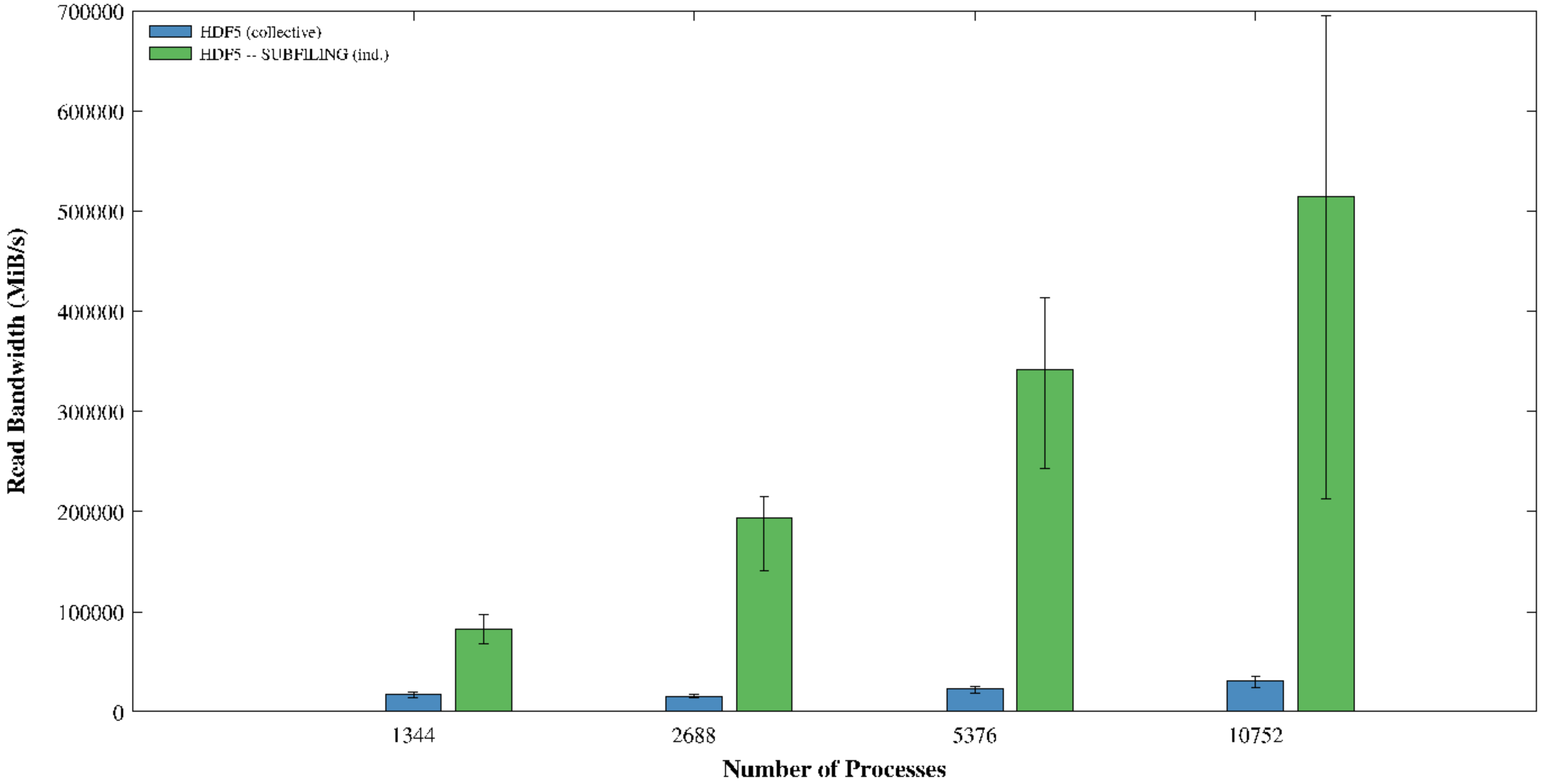


Performance Results -- Summit



- IOR, modified version for subfiling

Number of Ranks	File Size
1344	42GiB
2688	84 GiB
5376	168 GiB
10752	336 GiB



Availability and Requirements

- Initial version released in HDF5 1.13.2 release
  - Further development work has been merged to develop branch for HDF5 1.13.3 and 1.14.0 releases

- HDF5 must be built with parallel support enabled
  - Must enable subfiling when building HDF5. It’s not enabled by default

- C11 capable compiler support is required

- Requires MPI_Init_thread to be called by HDF5 application and requires MPI_THREAD_MULTIPLE level of threading support by MPI implementation

Questions?

- Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.