

# Combining multiple requests and nonblocking I/O

---

## Motivation

Array variables comprise the bulk of the data in a netCDF dataset, and for accesses to large regions of single array variables, PnetCDF attains very high performance. However, the current PnetCDF interface only allows access to one array variable per call. If an application instead accesses a large number of small-sized array variables, this interface limitation can cause significant performance degradation, because high end network and storage systems deliver much higher performance with larger request sizes. Moreover, the data for record variables are stored interleaved by record, and the contiguity information is lost, so the existing MPI-IO collective I/O optimization cannot help.

We provided a new mechanism for PnetCDF to combine multiple I/O operations for better I/O performance. This mechanism can be used in a new function that takes arguments for reading/writing multiple array variables, allowing application programmers to explicitly access multiple array variables in a single call. It can also be used in the implementation of nonblocking I/O functions, so that the combination is carried out implicitly, without changes to the application. Our performance evaluations, published in IASDS 2009, demonstrate significant improvement using well-known application benchmarks.

## Usage

There are two ways to use this feature:

**Explicit Method:** The caller explicitly accesses multiple variables at once. New routines added to the library (e.g. `ncmpi_mput_vara_all`, indicating multiple puts) take a list of variables to access.

The function calls for the explicit method look like this:

```
int
ncmpi_mput_vara_all(int ncid, /* in: dataset ID */
                     int nvrs, /* in: number of variables */
                     int varids[], /* in: [nvrs] list of variable IDs */
                     MPI_Offset* const starts[], /* in: [nvrs][ndims] list of start offsets */
                     MPI_Offset* const counts[], /* in: [nvrs][ndims] list of access counts */
                     void* const bufs[], /* in: [nvrs] list of buffer pointers */
                     MPI_Offset bufcounts[], /* in: [nvrs] list of buffer counts */
                     MPI_Datatype datatypes[]); /* in: [nvrs] MPI derived datatype describing bu
```

  

```
int
ncmpi_mget_vara_all(int ncid, /* in: dataset ID */
                     int nvrs, /* in: number of variables */
                     int varids[], /* in: [nvrs] list of variable IDs */
                     MPI_Offset* const starts[], /* in: [nvrs][ndims] list of start offsets */
                     MPI_Offset* const counts[], /* in: [nvrs][ndims] list of access counts */
                     void* bufs[], /* out: [nvrs] list of buffer pointers */
                     MPI_Offset bufcounts[], /* in: [nvrs] list of buffer counts */
                     MPI_Datatype datatypes[]); /* in: [nvrs] MPI derived datatype describing bu
```

Do note that we do not have Fortran bindings for these new routines in parallel-netcdf-1.1.0, but want very much to introduce Fortran bindings for this feature in a future release. In the meantime, Fortran codes can use the implicit method (below).

**Implicit Method** The library accesses multiple variables implicitly. Several variable accesses can be "scheduled" with the nonblocking routines. Then, when the application waits for completion of those accesses, the library will service them all in a single call.

Parallel-netcdf has had nonblocking routines in the C interface for some time. With this release we extend those routines to the Fortran interface.

In C:

```
int
ncmpi_iget_vara_float(int ncid,      /* in: dataset ID */
                      int varid,    /* in: variable ID */
                      const MPI_Offset start[], /* in: [ndims] start offsets */
                      const MPI_Offset count[],/* in: [ndims] access counts */
                      float *buf,   /* out: read buffer */
                      int *req_id); /* out: request ID */

int
ncmpi_iput_vara_float(int ncid,      /* in: dataset ID */
                      int varid,    /* in: variable ID */
                      const MPI_Offset start[], /* in: [ndims] start offsets */
                      const MPI_Offset count[],/* in: [ndims] access counts */
                      const float *buf, /* in: write buffer */
                      int *req_id); /* out: request ID */

int
ncmpi_wait(int ncid,      /* in: dataset ID */
           int num_reqs, /* in: number of requests */
           int req_ids[],/* in: [num_reqs] list of request IDs */
           int statuses[]);/* out: [num_reqs] list of request statuses */

int
ncmpi_wait_all(int ncid,      /* in: dataset ID */
               int num_reqs, /* in: number of requests */
               int req_ids[],/* in: [num_reqs] list of request IDs */
               int statuses[]);/* out: [num_reqs] list of request statuses */
```

In Fortran:

```
INTEGER FUNCTION nfmpi_iget_vara_real(ncid, varid, start, count, buf, req_id)
    INTEGER, INTENT(IN) :: ncid
    INTEGER, INTENT(IN) :: varid
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: start(*)
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: count(*)
    REAL, INTENT(OUT) :: buf(*)
    INTEGER, INTENT(OUT) :: req_id

INTEGER FUNCTION nfmpi_iput_vara_real(ncid, varid, start, count, buf, req_id)
    INTEGER, INTENT(IN) :: ncid
    INTEGER, INTENT(IN) :: varid
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: start(*)
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: count(*)
    REAL, INTENT(IN) :: buf(*)
    INTEGER, INTENT(OUT) :: req_id

INTEGER FUNCTION nfmpi_wait(ncid, num_reqs, req_ids, statuses)
    INTEGER, INTENT(IN) :: ncid
    INTEGER, INTENT(IN) :: num_reqs
```

```

INTEGER,                                     INTENT (IN)  :: req_ids(num_reqs)
INTEGER,                                     INTENT (IN)  :: statuses(num_reqs)

INTEGER FUNCTION nfmpi_wait_all(ncid, num_reqs, req_ids, statuses)
    INTEGER,                                         INTENT (IN)  :: ncid
    INTEGER,                                         INTENT (IN)  :: num_reqs
    INTEGER,                                         INTENT (IN)  :: req_ids(num_reqs)
    INTEGER,                                         INTENT (IN)  :: statuses(num_reqs)

```

The request posting calls, e.g. nfmpi\_iput\_vara\_float, can be called either collective or independent data mode. In the 'wait' and 'wait\_all' methods, the library actually batches up all outstanding nonblocking operations. In this way, we can carry out this optimization without needing a thread. Note that 'wait' must be called in independent data mode, while 'wait\_all' in collective.

## References

You can see examples of this new API in action in the test/mcoll\_perf directory:  
[https://trac.mcs.anl.gov/projects/parallel-netcdf/browser/trunk/test/mcoll\\_perf](https://trac.mcs.anl.gov/projects/parallel-netcdf/browser/trunk/test/mcoll_perf)

We wrote a paper describing the new APIs, their implementation, and some results:

Kui Gao, Wei-keng Liao, Alok Choudhary, Robert Ross, and Robert Latham. "Combining I/O Operations for Multiple Array Variables in Parallel NetCDF". In the *Proceedings of the Workshop on Interfaces and Architectures for Scientific Data Storage*, held in conjunction with the IEEE Cluster Conference, New Orleans, Louisiana, September 2009. [PDF](#)

We used this new API to achieve better checkpoint and plotfile I/O in the FLASH astrophysics code: Rob Latham, Chris Daley, Wei keng Liao, Kui Gao, Rob Ross, Anshu Dubey, and Alok Choudhary. "A case study for scientific I/O: improving the FLASH astrophysics code". *Computational Science & Discovery*, 5(1):015001, 2012. [online version](#)