I/O aggregation: implicit method (nonblocking APIs) and explicit method (multi-variable APIs)

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, most of the PnetCDF APIs that mimic netCDF’s only allow 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.

We provided a new mechanism for PnetCDF to aggregate/combine multiple I/O requests for better I/O performance. This mechanism is presented in two new sets of APIs: nonblocking APIs and multi-variable APIs. The former is used to implicitly to aggregate requests by posting multiple requests, each can be accessing to the same or a different variable, and then a single wait subroutine is used to combine the requests and commit to the file system. The latter is a new API family that takes arguments for reading/writing multiple array variables, allowing application programmers to explicitly access multiple array variables in a single call. Our performance evaluations, published in IASDS 2009 (reference below), demonstrate significant improvement using well-known application benchmarks.

Usage

Nonblocking APIs (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.

The term “nonblocking” here means the posting APIs are not blocked for waiting their I/O requests to complete. PnetCDF simply registers the nonblocking requests internally and immediately return the control to user program. There is no inter-process communication or I/O operation occurred in the posting call. The request aggregation and real I/O operation is carried out at the wait subroutines.

In C:

```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. ncmpi_iput_vara_float() or nfmpi_iput_vara_real(), 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 'ncmpi_wait_all()' must be called in collective data mode, while 'ncmpi_wait()' in independent mode.

Multi-variable APIs (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               nvars,        /* in:  number of variables */
                    int               varids[],     /* in:  [nvars] list of variable IDs */
                    MPI_Offset* const starts[],     /* in:  [nvars][ndims] list of start offsets */
                    MPI_Offset* const counts[]);     /* in:  [nvars][ndims] list of access counts */

Usage
Do note that we do not have Fortran bindings for these new routines in PnetCDF. Fortran codes should use the implicit method.

**Example programs**

- Implicit Method
  - C: nonblocking_write.c, block_cyclic.c, column_wise.c, pnetcdf-write-nb.c, pnetcdf-read-nb.c, pnetcdf-write-buffered.c
  - Fortran 77: nonblocking_write.f, block_cyclic.f, column_wise.f, pnetcdf-write-buffered77.f
  - Fortran 90: nonblocking_write.f90, block_cyclic.f90, column_wise.f90, pnetcdf-write-bufferedf.f90

- Explicit Method mput.c

**References**

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 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](#)