

# Parallel and Large-scale Simulation Enhancements to CGNS

M. Scot Breitenfeld, The HDF Group

October 15, 2014

## 1 Overview of changes introduced in the HDF5\_Parallel branch

Many of the changes discussed in the following sections address the currently (as of version 3.2.1) underperforming parallel capabilities of the CGNS library. For example, the CGNS function `cgp_open`, which opens a CGNS file for processing, has substantially increasing execution time as the number of processes is increased, Fig. 1 (trunk). The current improvement for `cgp_open` is substantial at 100-1000 times faster (branch) than the previous implementation (trunk). In fact, for runs with the largest number of processes ( $>1024$ ) the batch job had a time limit of 5 minutes and not all the processes had completed `cgp_open` before this limit was reached. Obviously, the previous implementation of `cgp_open` is a lot worse than reported in the figure.

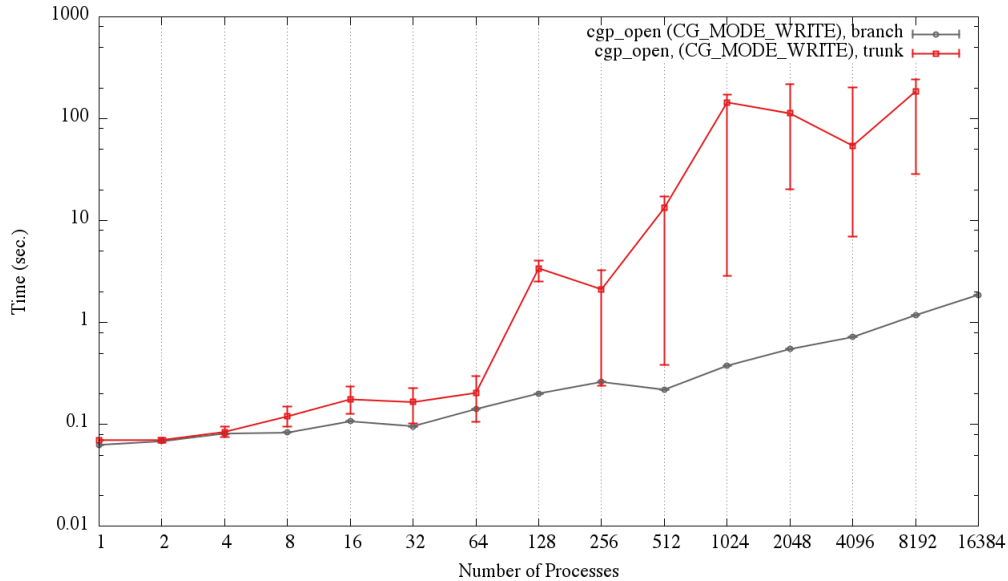


Figure 1: Time for completion of `cgp_open` in write mode for the original implementation (trunk) and the current implementation (branch), the error bars correspond to the minimum and maximum time over all the processes that had completed before the batch job time limit was reached.

Section 2 lists changes to CGNS that could effect the end user and introduces new functions and specifications. Fortran programmers should take notice of Section 2.2 which highlights changes introduced for better interoperability with the C CGNS library. Section 3 gives example installation guides for GPFS and Lustre hardware. Items listed in blue affect compatibility of older codes when using CGNS v3.2.2. Known problems are highlighted in red. The branch can be checked out from sourceforge at,

[http://svn.code.sf.net/p/cgns/code/cgns/branches/HDF5\\_Parallel](http://svn.code.sf.net/p/cgns/code/cgns/branches/HDF5_Parallel)

## 2 General behavior changes and new recommendations for parallel performance

- The flush functions should not be used. Writing and reading immediately avoids IO contention occurring when flush is being used.
- The parallel routines are meant for parallel file systems (GPFS or Lustre).
- The default parallel input/output mode was changed from CGP\_INDEPENDENT to CGP\_COLLECTIVE.
- An extra argument for passing MPI info to the CGNS library was added to cgp\_pio\_mode.

### C

```
int cgp_pio_mode(CGNS_ENUMT(PIOmode_t) mode, MPI_Info info)
```

### Fortran

```
CALL cgp_pio_mode_f(mode, comm_info, ierr)
INTEGER(KIND(CGP_COLLECTIVE)) :: mode ! Use parameters CGP_INDEPENDENT or CGP_COLLECTIVE
INTEGER :: comm_info
INTEGER :: ierr
```

- Functions for parallel reading and writing multi-component datasets using a single call was introduced. The new APIs use new capabilities introduced in version 1.8.\* (currently not released as of 10.13.2014) of the HDF5 library. The new APIs pack multiple datasets into a single buffer and the underlying MPI IO completes the IO request using just one call. The availability of the new functions in the HDF5 library is checked at compile time. The current limitation (due to MPI) is that the size of the sum of the datasets must be less than 2GB. Example usage can be found in benchmark\_hdf5.c and benchmark\_hdf5\_f90.F90 in ptests.

### C

```
int cgp_coord_multi_read_data(int fn, int B, int Z, int *C,
                             const cgsized_t *rmin, const cgsized_t *rmax,
                             void *coordsX, void *coordsY, void *coordsZ);

int cgp_coord_multi_write_data(int fn, int B, int Z, int *C,
                              const cgsized_t *rmin, const cgsized_t *rmax,
                              const void *coordsX, const void *coordsY, const void *coordsZ);

int cgp_field_multi_read_data(int fn, int B, int Z, int S, int *F,
                             const cgsized_t *rmin, const cgsized_t *rmax,
                             int nsets, ...);
/* ... nsets of variable arguments, *solution_array, corresponding to the order given by F */

int cgp_field_multi_write_data(int fn, int B, int Z, int S, int *F,
                              const cgsized_t *rmin, const cgsized_t *rmax,
                              int nsets, ...);
/* ... nsets of variable arguments, *solution_array, corresponding to the order given by F */

int cgp_array_multi_write_data(int fn, int *A, const cgsized_t *rmin, const cgsized_t *rmax,
                              int nsets, ...);
/* ... nsets of variable arguments, *field_array, corresponding to the order given by F */
```

```

int cgp_array_multi_read_data(int fn, int *A, const cgsizet *rmin, const cgsizet *rmax,
                             int nsets, ...);
/* ... nsets of variable arguments, *field_array, corresponding to the order given by F */

```

## Fortran

```

CALL cgp_coord_multi_read_data_f(fn, B, Z, C, rmin, rmax, coordsX, coordsY, coordsZ, ier)
INTEGER :: fn
INTEGER :: B
INTEGER :: Z
INTEGER :: C
INTEGER(CG_SIZE_T) :: rmin
INTEGER(CG_SIZE_T) :: rmax
REAL :: coordsX, coordsY, coordsZ
INTEGER :: ier

```

```

CALL cgp_coord_multi_write_data_f(fn, B, Z, C, rmin, rmax, coordsX, coordsY, coordsZ, ier)
INTEGER :: fn
INTEGER :: B
INTEGER :: Z
INTEGER :: C
INTEGER(CG_SIZE_T) :: rmin
INTEGER(CG_SIZE_T) :: rmax
REAL :: coordsX, coordsY, coordsZ
INTEGER :: ier

```

```

CALL cgp_field_multi_write_data_f(fn, B, Z, S, F, rmin, rmax, ier, nsets, ...)
INTEGER :: fn
INTEGER :: B
INTEGER :: Z
INTEGER :: C
INTEGER(CG_SIZE_T) :: rmin
INTEGER(CG_SIZE_T) :: rmax
INTEGER :: ier
INTEGER :: nsets
... REAL, DIMENSION(*) :: field_array ! entered nsets times

```

```

CALL cgp_field_multi_read_data_f(fn, B, Z, S, F, rmin, rmax, ier, nsets, ...)
INTEGER :: fn
INTEGER :: B
INTEGER :: Z
INTEGER :: C
INTEGER(CG_SIZE_T) :: rmin
INTEGER(CG_SIZE_T) :: rmax
INTEGER :: ier
INTEGER :: nsets
... REAL, DIMENSION(*) :: field_array ! entered nsets times

```

```

CALL cgp_array_multi_write_data_f(fn, B, Z, S, F, rmin, rmax, ier, nsets, ...)
INTEGER :: fn
INTEGER :: B
INTEGER :: Z
INTEGER :: C
INTEGER(CG_SIZE_T) :: rmin
INTEGER(CG_SIZE_T) :: rmax
INTEGER :: ier
INTEGER :: nsets
... REAL, DIMENSION(*) :: data_array ! entered nsets times

CALL cgp_array_multi_read_data_f(fn, B, Z, S, F, rmin, rmax, ier, nsets, ...)
INTEGER :: fn
INTEGER :: B
INTEGER :: Z
INTEGER :: C
INTEGER(CG_SIZE_T) :: rmin
INTEGER(CG_SIZE_T) :: rmax
INTEGER :: ier
INTEGER :: nsets
... REAL, DIMENSION(*) :: data_array ! entered nsets times

```

## 2.1 New C changes

- A new parallel example benchmark program, `benchmark_hdf5.c`, was added to directory `ptests`.

## 2.2 New Fortran changes

All users are **strongly** encouraged to use a Fortran 2003 standard compliant compiler. Using a Fortran 2003 compiler guarantees interoperability with the C APIs via the `ISO_C_BINDING` module. Many changes were added to the CGNS library in order to take full advantage of the interoperability offered by the `ISO_C_BINDING` module.

1. Configure was changed to check if the Fortran compiler is Fortran 2003 compliant. If it is then the features of `ISO_C_BINDING` will be used.
2. The predefined CGNS constant parameters data types were changed from `INTEGER` to `ENUM`, `BIND(C)` for better C interoperability. The users should use the predefined constants whenever possible and not the numerical value represented by the constants.
3. *INCLUDE* “`cgslib_h`” was changed in favor of using a module, `USE CGNS`.
  - (a) This allows defining a `KIND` type for integers instead of the current way of using the preprocessor dependent `cgsize_t`.
  - (b) Backward compatibility might be added before the merge to the trunk.
4. The user should be sure to declare the arguments declared *int* in the C APIs as `INTEGER` in Fortran. The ONLY Fortran arguments declared as type `cgsize_t` should be the arguments which are also declared `cgsize_t` in the C APIs. This is very important when building with option `-enable-64bit`.
5. Assuming the rules in step 4 were followed, users should not need to use parameter `CG_BUILD_64BIT` since Fortran’s `cgsize_t` is now guaranteed to match C’s `cgsize_t`.

6. Fortran programs defining CGNS data types with a default INTEGER size of 8 bytes are not currently compatible. This is independent of whether or not `-enable-64bit` is being used. For clarification, using `-enable-64bit` allows for data types (i.e. those declared as `cgsizet`) to be able to store values which are too large to be stored as 4 byte integers (i.e. numbers greater than 2,147,483,647). It is not necessary, or advisable, to have CGNS INTEGER types (types declared `int` in C) to be 8 bytes; the variables declared as `cgsizet` will automatically handle data types that can not be stored as 4 byte integers when `-enable-64bit` is being used.
  - (a) CGNS developer's note: A new C data type, `cgint_f`, was introduced to be interpretable with the C type `int`. In order to allow for default 8 byte integers in Fortran: (1) The C API wrappers in `cg_ftoc.c` need to be changed from `cgsizet` to `cgint_f` everywhere the C argument is declared as an `int` in C, (2) configure needs to detect what size the default integer is in Fortran and find the corresponding size in C in order to set the correct size of `cgint_f`.
7. Two new benchmarking programs were introduced in directory `ptests`:
  - (a) `benchmarking_hdf5_f90.F90` uses the conventional Fortran wrappers.
  - (b) `benchmarking_hdf5_f03.F90` calls the C APIs directly, no Fortran wrappers are used.
8. A new Fortran API was added for determining the CGNS data type of a variable which is interoperable with the C data type.

```
Function cg_get_type(var)
  type, INTENT(IN) :: var
  INTEGER(KIND(enumvar)) :: cg_get_type
```

An example of using the new function to automatically specify the CGNS type corresponding to the Fortran data type is,

```
INTEGER, DIMENSION(1:10) :: Array_i

CALL cgp_array_write_f("ArrayI",cg_get_type(Array_i(1)),1,INT(nijk(1),cgsizet),Ai, err)
```

## 3 Parallel installation instructions

Two parallel files systems were investigated: GPFS (mira, Argonne National Laboratory) and Lustre (Pleiades NASA). The following descriptions were for those systems, but the overall procedure should be similar on different machines of the same type. Example build scripts for these systems can be found in `src/sampleScripts` of the CGNS source code. They include scripts for building zlib, hdf5 (assuming the user does not already have them installed system wide) and a script for building CGNS. All the scripts use autotools; **cmake remains untested**. The next few examples assume all the needed packages are in `${HOME}/packages` and all the build scripts are placed in `${HOME}/packages`. This information can also be found in the `README.txt` in the scripts directory.

### 3.1 Building on IBM Blue Gene (GPFS)

1. Building zlib from source: Download and extract the zlib source: <http://www.zlib.net/>
  - (a) `cd` into the top level zlib source directory.
  - (b) modify and run the script: `../build_zlib`
2. Building hdf5 from source

- (a) From the top level of the hdf5 library, change the `${HOME}/packages` to where zlib was installed in STEP 1.
- (b) `../build_hdf5 -without-pthread -disable-shared -enable-parallel -enable-production \ -enable-fortran -enable-fortran2003 \ -disable-stream-vfd -disable-direct-vfd \ -with-zlib=${HOME}/packages/zlib-1.2.8/lib -prefix=${HOME}/packages/phdf5-trunk`

where prefix is set for where the hdf5 library will get installed. There should be no need to modify the script.

### 3. Building cgns from source:

- (a) cd into the cgns/src directory
- (b) modify and run: `<pathto>/build_cgns`
- (c) make
- (d) To make the tests: `cd ptests; make; make tests`

### 4. Important parameters for good performance on GPFS:

- (a) The environment variable `BGLOCKLESSMPIO_F_TYPE=0x47504653` should be set. For example, this can be set in a batch job using `qsub -env BGLOCKLESSMPIO_F_TYPE=0x47504653`

## 3.2 Building on SGI (Lustre)

### 1. Building zlib from source: Download and extract the zlib source: <http://www.zlib.net/>

- (a) cd into the top level zlib source directory.
- (b) modify and run the script: `../build_zlib`

### 2. Building hdf5 from source:

- (a) From the top level of the hdf5 library, change the `${HOME}/packages` to where zlib was installed in STEP 1.
- (b) `../build_hdf5`

### 3. Building cgns from source:

- (a) cd into the cgns/src directory
- (b) modify and run: `<pathto>/build_cgns`
- (c) make
- (d) To make the tests: `cd ptests; make; make tests`

### 4. Important parameters for good performance:

- (a) The Lustre parameters have not been fully tested.
- (b) On Pleiades, `lfs setstripe -c 64 -s 0 /nobackupp8/<dir>`, has shown good performance.

## Acknowledgments

This work was funded by NASA through the CGNS project, contract #NNL14AB41T. Additionally, the computational studies done on cetus and mira at Argonne National Laboratory used allocation time through the ExaHDF5 project, DOE contract #DE-AC02-05CH11231. Computational studies on the Lustre file system was done on Pleiades through NASA's High-End Computer Capabilities Project.