benchio

1.0.0

Generated by Doxygen 1.11.0

Chapter 1

C-based benchio

Simple C parallel IO benchmark for teaching and benchmarking purposes.

This is a ported version of the benchio parallel IO benchmark, which was originally developed in Fortran by the EPCC: https://github.com/davidhenty/benchio.

1.1 Installing benchio

Note that, before running the benchmark, you *must* manually set the striping on the three directories `unstriped`, `striped` and `fullstriped`.

If you are running Lustre (for example on Cirrus and ARCHER2), then these are the instructions to do so:

- Set `unstriped` to have a single stripe: `lfs setstripe -c 1 unstriped`
- Set `fullstriped` to use the maximum number of stripes: `lfs setstripe -c -1 fullstriped`
- Set `striped` to use an intermediate number of stripes, e.g. for 4 stripes: `lfs setstripe -c 4 striped`

If you are running some other filesystem, then check the user guide for that system.

1.1.1 ARCHER2 Makefile

A sample makefile is available for ARCHER2. You must first load the required modules for ADIOS2 to install; instructions are included inside the Makefile. You can then simply run `make -f Makefile_ARCHER2` to compile. If you desire to compile without ADIOS2, then run `NOADIOS=true make -f Makefile_ARCHER2`.

1.1.2 Cirrus Makefile

A sample makefile is available for Cirrus. Load MPI (e.g. `module load mpt`) and then use `make -f Makefile_Cirrus` to compile. Note that this will compile without ADIOS2 by default. If you have installed ADIOS2 locally, then you can compile with ADIOS2 using `USEADIOS=true make -f Makefile_Cirrus`

1.1.3 Other Systems

You will have to alter one of the existing makefiles to suit your needs. Note that benchio is designed such that if the macro `NOADIOS` is defined, then ADIOS2-specific code is excluded from the compilation.

1.2 Running benchio

To run benchio, you must specify the dimensions of the 3D dataset to write through the `-n1`, `-n2` and `-n3` flags. Additionally, you must specify whether the sizes provided are to apply per process, or globally, the `-sc (local|global)` flag.

For example, to run using a 256 x 256 x 256 data array on every process (i.e. weak scaling):
```
benchio -n1 256 -n2 256 -n3 256 -sc local
```

In this case, the total file size will scale with the number of processes. If run on 8 processes then the total file size would be 1 GiB.

To run using a 256 x 256 x 256 global array (i.e. strong scaling):
```
benchio -n1 256 -n2 256 -n3 256 -sc global
```

In this case, the file size will be 128 MiB regardless of the number of processes.

By default, benchio only measures write time. To read the file back immediately after reading and record the time taken, use the `-r` flag.

A 3D cartesian topology p1 x p2 x p3 is created with dimensions suggested by `MPI_Dims_create()` to create a global 3D array of size l1 x l2 x l3 where l1 = p1 x n1 etc. The entries of the distributed IO array are set to globally unique values 1, 2, ... l1xl2xl3 using the normal C ordering.

The code can use seven IO methods, and for each of them can use up to three directories with different stripings. At the moment, the C version of benchio only supports the `serial/proc/node/mpiio/adios` options, and will reject the other options.

All files are deleted immediately after being written to avoid excess disk usage.

The full set of options is:
```
benchio -n1 (size) -n2 (size) -n3 (size) (--scale|-sc) (local|global)
        [--mode|-m] [serial] [proc] [node] [mpiio] [hdf5] [netcdf] [adios]
        [--stripe|-st] [unstriped] [striped] [fullstriped]
        [--read|-r]
```

Additionally, `benchio --help` (or `benchio -h`) can be used to get more information on each option.

If `--mode` is not specified, then all the IO modes are used. Similarly, if `--stripe` is not specified, then the program will use all striping methods.

1. `serial`: Serial IO from one controller process to a single file `serial.dat` using C binary unformatted write with `fopen(..., "wb");`
   (a) `proc`: File-per-process with multiple serial IO to *P* files `rankXXXXXX.dat` using C binary unformatted write with `fopen(.., "wb");`
   (b) `node`: File-per-node with multiple serial IO to *Nnode* files `nodeXXXXXX.dat` using C binary unformatted write with `fopen(.., "wb");`
   (c) `mpiio`: MPI-IO collective IO to a single file `mpiio.dat` using native (i.e. binary) format
   (d) `hdf5`: HDF5 collective IO to a single file `hdf5.dat`
   (e) `netcdf`: NetCDF collective IO to a single file `netcdf.dat`
   (f) `adios`: ADIOS2 collective IO to a directory/file `adios.dat`
      - ADIOS2 aggregator settings can be changed in the `adios2_config.xml` file

   Note that the serial part is designed to give a baseline IO rate. For simplicity, and to ensure we write the same amount of data as for the parallel methods, rank 0 writes out its own local array `size` times in succession. Unlike the parallel IO formats, the contents of the file will therefore *not* be a linearly increasing set of values 1, 2, 3, ..., l1xl2xl3.

1.3 Debug Mode

The C version of benchio includes the ability to check correctness of the dataset that is written to or read from disk. To enable this, compile the software with the DEBUG_MODE pragma set to true. The sample makefiles also include convenience features: `DEBUG_MODE=true make -f Makefile_ARCHER2`.

1.4 Known Issues

Known Issues in Release 1.0.0:

- ADIOS2 HDF5 mode is disabled in this version of benchio, because it does not work properly
- ADIOS2 read based benchmark is heavily distorted by caching effects

1.5 Documentation

See the subfolder "Doxygen" for a PDF documentation of the software. The file "Doxyfile" is also provided, should you wish to generate your own version of the documentation.

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Class Documentation

4.1 argument Struct Reference

Helpful struct to handle the user input passed to the program.

`#include <benchio.h>`

Public Attributes

- char * long_arg
- char * short_arg
- bool standalone
- bool optional
- bool found
- bool complete

4.1.1 Detailed Description

Helpful struct to handle the user input passed to the program.

4.1.2 Member Data Documentation

4.1.2.1 complete

`bool argument::complete`

Parameter used when processing arguments to indicate if the argument should take no more input

4.1.2.2 found

`bool argument::found`

Parameter used when processing arguments to indicate if the argument was found or not

4.1.2.3 long_arg

```
char* argument::long_arg
```

The long version of the argument

4.1.2.4 optional

```
bool argument::optional
```

Whether or not to go ahead if the argument was not provided

4.1.2.5 short_arg

```
char* argument::short_arg
```

The short version of the argument

4.1.2.6 standalone

```
bool argument::standalone
```

Whether or not the argument needs more input than itself

The documentation for this struct was generated from the following file:

- benchio.h

Chapter 5

File Documentation

5.1 adios2.c File Reference

File for ADIOS2 related code in benchio.

```
#include "benchio.h"
#include <adios2_c.h>
```

Functions

- void adios2_write (char const *file_name, double ***io_data, double *local_sizes, double *global_sizes, MPI_Comm cartesian_comm)

  Perform an ADIOS2 write of the global array to file.
- void adios2_read (char const *file_name, double ***io_data, double *local_sizes, double *global_sizes, MPI_Comm cartesian_comm)

  Perform an ADIOS2 read of the global array from file.
- void adios2_verify (char *file_name, double *global_sizes, MPI_Comm communicator)

  Read data which has been written with ADIOS2 back and verify its correctness.
- bool get_adios2_io_mode (MPI_Comm io_comm, enum io_mode *adios_io_mode)

  Retreive the current ADIOS2 IO mode in use.
- bool adios2_native_cleanup (char const *file_name, enum io_mode io_mode)

  Clean up the files written by the native ADIOS modes.

Variables

- char *const **bp3_file_names** [ ] = {".bp", ".bp.dir/adios.dat.bp.%d", ".bp.dir/profiling.json"}

  File names of each ADIOS2 bp3 native binary format mode, to delete when cleaning up.
- char *const **bp4_file_names** [ ] = {"/data.%d", "/md.%d", "/md.idx", "/profiling.json"}

  File names of each ADIOS2 bp4 native binary format mode, to delete when cleaning up.
- char *const **bp5_file_names** [ ] = {"/data.%d", "/md.%d", "/mmd.%d", "/md.idx", "/profiling.json"}

  File names of each ADIOS2 bp5 native binary format mode, to delete when cleaning up.

5.1.1 Detailed Description

File for ADIOS2 related code in benchio.

Contains code relating to reading, writing, verifying correctness of, and deleting ADIOS2 files.

5.1.2 LICENSE

This software is released under the MIT License.

5.1.3 Function Documentation

5.1.3.1 adios2_native_cleanup()

```
bool adios2_native_cleanup (
            char const * file_name,
            enum io_mode io_mode io_mode )
```

Clean up the files written by the native ADIOS modes.

Remove the files written by ADIOS2 modes bp3, bp4, or bp5. Since ADIOS2 writes a folder, with the files of the folder varying depending on which mode is used, and the aggregator count, this removal uses 'brute force' to accomplish this; see details.

Parameters

in	*file_name*	The name of the file or resource to clean up.
in	*io_mode*	The ADIOS2 IO mode being used

Returns

Returns `true` if cleanup was successful, `false` otherwise.

This is a bit of a workaround for deletion with ADIOS, which generates a set of files for its native binary formats. The goal is to have a platform-independent solution to delete whatever file was generated. Unfortunately not many good options exist, so solution here is to delete files one-by-one depending on mode used. An additional complication is that many numbered files are sometimes generated, e.g. data.0, data.1, ... This is solved here with a do-while loop, which just keeps incrementing a counter and deleting as long as it is successful.

5.1.3.2 adios2_read()

```
void adios2_read (
            char const * file_name,
            double *** io_data,
            double * local_sizes,
            double * global_sizes,
            MPI_Comm cartesian_comm)
```

Perform an ADIOS2 read of the global array from file.

This function reads data from a file using the ADIOS2 library. It reads the data into the `io_data` array using the specified file name, local and global sizes, and MPI communicator.

Parameters

in	*file_name*	Name of the ADIOS2 file (folder) to read from
in,out	*io_data*	The pre-allocated current process' 3D array to fill with data from file
in	*local_sizes*	An array of values indicating the per-process array dimensions
in	*global_sizes*	An array of values indicating the global array dimensions
in	*cartesian_comm*	An MPI communicator with a cartesian topology

5.1.3.3 adios2_verify()

```
void adios2_verify (
            char * file_name,
            double * global_sizes,
            MPI_Comm communicator)
```

Read data which has been written with ADIOS2 back and verify its correctness.

Use a single process to read the ADIOS2 data which was written to file through the benchmark, loading it into a 1D array and verifying that it forms the correct pattern (a series of incrementing double-precision values starting from 1 and ending at global_sizes[0]∗global_sizes[1]∗global_sizes[2])

Parameters

in	*file_name*	Name of the ADIOS2 file (folder) to read from
in	*global_sizes*	An array of values indicating the global array dimensions
in	*communicator*	The global MPI communicator

5.1.3.4 adios2_write()

```
void adios2_write (
            char const * file_name,
            double *** io_data,
            double * local_sizes,
            double * global_sizes,
            MPI_Comm cartesian_comm)
```

Perform an ADIOS2 write of the global array to file.

Write the global array, stored as a set of local arrays in io_data, to the specified file, using ADIOS2. Perform error error checking throughout to report any issues encountered.

Parameters

in	*file_name*	Name of the ADIOS2 file (folder) to create
in	*io_data*	The current process' 3D array of data forming its part of the global array
in	*local_sizes*	An array of values indicating the per-process array dimensions
in	*global_sizes*	An array of values indicating the global array dimensions
in	*cartesian_comm*	An MPI communicator with a cartesian topology

5.1.3.5 get_adios2_io_mode()

```
bool get_adios2_io_mode (
            MPI_Comm io_comm,
            enum io_mode * adios_io_mode)
```

Retreive the current ADIOS2 IO mode in use.

Starts ADIOS2 from the configuration file and reads the IO engine to use

Parameters

in	*io_comm*	The global MPI communicator
out	*adios_io_mode*	Pointer to enum io_mode where to store the retreived IO mode

Returns

Returns `true` if successfully read the IO mode of ADIOS2, `false` otherwise.

5.2 benchio.c File Reference

Program starting point for benchio.

```
#include <ctype.h>
#include <time.h>
#include <string.h>
#include "benchio.h"
```

Functions

- int main (int argc, char **argv)

  Main benchio starting point.
- void main_benchmark_loop (MPI_Comm cartesian_comm, MPI_Comm node_comm, bool *use_stripe_method, bool *use_io_method, double *local_sizes, double *global_sizes, double global_size_gib, bool read_benchmark, int node_num, enum io_mode adios_io_mode, int *coords, double ***io_data)

  Start the benchmark, looping through each step according to configuration, and report results.
- bool run_read_benchmark (char *file_name, double *local_sizes, double *global_sizes, MPI_Comm io_comm, int my_rank, int io, double global_size_gib, enum io_mode io_mode, MPI_Comm cartesian_comm, int *coords)

  Run the read-based benchmark according to specified configuration.
- bool run_write_benchmark (char *file_name, double ***io_data, double *local_sizes, double *global_sizes, MPI_Comm io_comm, int my_rank, int io, double global_size_gib, enum io_mode io_mode)

  Run the write-based benchmark according to specified configuration.
- void populate_io_data (double *local_sizes, int *coords, double *global_sizes, double ***io_data)

  Fill the current process' array with data consistent with the global array.
- bool process_args (int argc, char **argv, int my_rank, int *sizes, bool *global_flag, bool *use_io_method, bool *use_stripe_method, bool *read_benchmark, bool *user_wants_help)

  Parse command-line arguments provided by user.
- void setup_nodes (MPI_Comm communicator, int original_rank, MPI_Comm *node_comm, int *node_number)

  Set up communicators reflecting the nodal environment currently in use.

Variables

- char const ∗const **io_method_names** [ ] = {"serial", "proc", "node", "mpiio", "hdf5", "netcdf", "adios"}

  The names of each of the io methods, in a lowercase format. Used for file creation etc.
- char const ∗const **io_formatted_names** [ ] = {"Serial", "Proc", "Node", "MPI-IO", "HDF5", "NetCDF", "Adios2"}

  The names of each of the io methods, in a nice-to-print format. Used for user output.
- char const ∗const **stripe_method_names** [ ] = {"unstriped", "striped", "fullstriped"}

  The names of each of the stripe methods, used for file creation etc.
- struct argument expected_arguments [ ]

  The legal arguments to accept. Used in process_args(...).

5.2.1 Detailed Description

Program starting point for benchio.

Parse user input and determine the configuration to be used, then run the program main benchmark loop accordingly.

5.2.2 LICENSE

This software is released under the MIT License.

5.2.3 Function Documentation

5.2.3.1 main()

```
int main (
            int argc,
            char ** argv)
```

Main benchio starting point.

Process arguments, identify the environment, and start the main loop of the benchmark.

Parameters

argc	Number of arguments from user
argv	Argument array from user

Returns

EXIT_SUCCESS on successful completion, otherwise EXIT_FAILURE

5.2.3.2 main_benchmark_loop()

```
void main_benchmark_loop (
            MPI_Comm cartesian_comm,
            MPI_Comm node_comm,
            bool * use_stripe_method,
            bool * use_io_method,
            double * local_sizes,
            double * global_sizes,
            double global_size_gib,
            bool read_benchmark,
            int node_num,
            enum io_mode adios_io_mode,
            int * coords,
            double *** io_data)
```

Start the benchmark, looping through each step according to configuration, and report results.

Take some identified set of settings, such as the IO methods to use, the stripings to use, and whether or not to run a read-back benchmark, and run the benchmark accordingly.

Parameters

in	*cartesian_comm*	An MPI communicator with a cartesian topology
in	*node_comm*	An MPI communicator specific to the current process' node
in	*use_stripe_method*	An array of booleans indicating whether or not to use each striping method
in	*use_io_method*	An array of booleans indicating whether or not to use each io method
in	*local_sizes*	An array of values indicating the per-process array dimensions
in	*global_sizes*	An array of values indicating the global array dimensions
in	*global_size_gib*	The total global array size in GiB.
in	*read_benchmark*	Boolean flag indicating whether or not to also perform a read benchmark
in	*node_num*	The rank of the process' current node in the node-spanning communicator
in	*adios_io_mode*	Which underlying IO mode ADIOS is using, if applicable
in	*coords*	The coordinates of the current process in cartesian_comm
in	*io_data*	The current process' 3D array of data forming its part of the global array

5.2.3.3 populate_io_data()

```
void populate_io_data (
            double * local_sizes,
            int * coords,
            double * global_sizes,
            double *** io_data)
```

Fill the current process' array with data consistent with the global array.

Fill the current process' array with data such that, in the cartesian grid of processes, it forms part of a series of incrementing double-precision values from 1 to global_sizes[0]∗global_sizes[1]∗global_sizes[2].

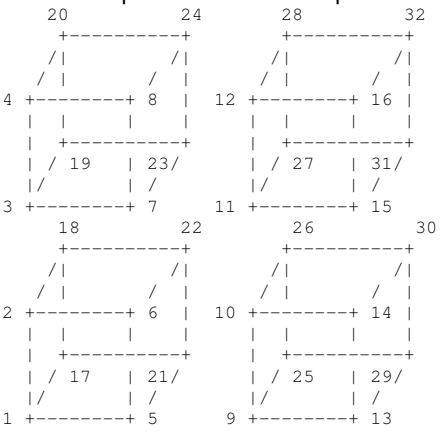
Parameters

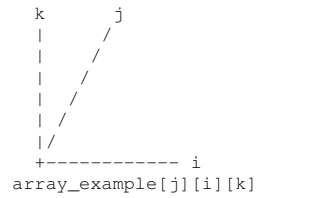
in	*local_sizes*	The array of sizes on the current process

in	*coords*	The array of coordinates in the cartesian communuicator
in	*global_sizes*	The array of sizes of the global array
in,out	*io_data*	Pointer to a pre-allocated 3D array to populate

Populate the array of the current process. Each element contains a globally unique double-precision value between 1 and global_sizes[0]∗global_sizes[1]∗global_sizes[2]. Count increments by one for each element.

Example ordering on a (2, 2, 1) decomposition with 2x2x2 local data. Each cube is a process' own array. P0 is bottom-left process. With more processes, extend pattern...






```
array_example[j][i][k]

Notice that MPI structures its process grid as follows:
mpi_dims_example[i][k][j]
```

5.2.3.4 process_args()

```
bool process_args (
             int argc,
             char ** argv,
             int my_rank,
             int * sizes,
             bool * global_flag,
             bool * use_io_method,
             bool * use_stripe_method,
             bool * read_benchmark,
             bool * user_wants_help)
```

Parse command-line arguments provided by user.

The the command-line arguments provided by the user and validate them. Print useful debug to user and return if some of their input was invalid, otherwise quietly fill the parameters with the specified input.

Parameters

in	*argc*	Number of arguments from user

Parameters

in	*argv*	Argument array from user
in	*my_rank*	Current process rank in the global MPI communicator
out	*sizes*	Array of size DIMENSIONS to store the user input for dimensions
out	*global_flag*	Pointer to boolean to store flag of whether to use local or global scaling
out	*use_io_method*	Array of size IO_METHOD_COUNT to store boolean values indicating whether to use each IO method
out	*use_stripe_method*	Array of size STRIPE_TYPE_COUNT to store boolean values indicating whether to use each striping method
out	*read_benchmark*	Pointer to a boolean to store flag of whether to also perform read-based benchmark
out	*user_wants_help*	Pointer to a boolean to store flag of whether user asked for help through –help or -h flags

Returns

Return `true` if OK to continue benchmark, otherwise `false`

5.2.3.5 run_read_benchmark()

```
bool run_read_benchmark (
            char * file_name,
            double * local_sizes,
            double * global_sizes,
            MPI_Comm io_comm,
            int my_rank,
            int io,
            double global_size_gib,
            enum io_mode io_mode io_mode ,
            MPI_Comm cartesian_comm,
            int * coords)
```

Run the read-based benchmark according to specified configuration.

Run the read-based benchmark and report results. If DEBUG_MODE is enabled, this function also validates the data which was read back in for correctness.

Parameters

in	*file_name*	Name of the file to read from
in	*local_sizes*	An array of values indicating the per-process array dimensions
in	*global_sizes*	An array of values indicating the global array dimensions
in	*io_comm*	The MPI communicator context to perform the read in
in	*my_rank*	The rank of the current process in the global communicator
in	*io*	An integer indicating which IO mode to read using; see benchio.h compile-time constants
in	*global_size_gib*	The size of the global array in GiB.
in	*io_mode*	Which underlying IO mode is in use (ADIOS2 has multiple engines)
in	*cartesian_comm*	The cartesian-topology communicator which was used to populate data
in	*coords*	Coordinates of current process in cartesian_comm, size DIMENSIONS

Returns

Returns `true` if benchmark completed normally, `false` otherwise.

5.2.3.6 run_write_benchmark()

```
bool run_write_benchmark (
            char * file_name,
            double *** io_data,
            double * local_sizes,
            double * global_sizes,
            MPI_Comm io_comm,
            int my_rank,
            int io,
            double global_size_gib,
            enum io_mode io_mode io_mode )
```

Run the write-based benchmark according to specified configuration.

Run the write-based benchmark and report results. If DEBUG_MODE is enabled, this function also validates the data which was written by reading it back in and checking it for correctness.

Parameters

in	*file_name*	Name of the file to read from
in	*io_data*	The current process' 3D array of data forming its part of the global array
in	*local_sizes*	An array of values indicating the per-process array dimensions
in	*global_sizes*	An array of values indicating the global array dimensions
in	*io_comm*	The MPI communicator context to perform the write in
in	*my_rank*	The rank of the current process in the global communicator
in	*io*	An integer indicating which IO mode to read using; see benchio.h compile-time constants
in	*global_size_gib*	The size of the global array in GiB.
in	*io_mode*	Which underlying IO mode is in use (ADIOS2 has multiple engines)

Returns

Returns `true` if benchmark completed normally, `false` otherwise.

5.2.3.7 setup_nodes()

```
void setup_nodes (
            MPI_Comm communicator,
            int original_rank,
            MPI_Comm * node_comm,
            int * node_number)
```

Set up communicators reflecting the nodal environment currently in use.

Create node-spanning communicators and identify the node bosses (rank 0s) of each node. Also identify and print out the name of each node.

Parameters

in	*communicator*	An MPI communicator with all processes in it
in	*original_rank*	Rank of current process within `communicator`
out	*node_comm*	Pointer to communicator to fill with a communicator spanning the current node
out	*node_number*	Pointer to integer which will be filled with the node number of the current process

5.2.4 Variable Documentation

5.2.4.1 expected_arguments

```
struct argument expected_arguments[]
```

Initial value:

```
= {
    {"n1", "n1", false, false, false, false},
    {"n2", "n2", false, false, false, false},
    {"n3", "n3", false, false, false, false},
    {"scale", "sc", false, false, false, false},
    {"mode", "m", false, true, false, false},
    {"stripe", "st", false, true, false, false},
    {"read", "r", true, true, false, false},
    {"help", "h", true, true, false, false}}
```

The legal arguments to accept. Used in process_args(...).

See the struct definition in benchio.h for explanation of parameters.

5.3 benchio.h File Reference

Header file for benchio program.

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
```

Classes

- struct argument

  Helpful struct to handle the user input passed to the program.

Macros

- #define **RANK_ZERO** 0

  Rank 0 constant included for clarity.
- #define **DIMENSIONS** 3

  Dimensions of processes and dataset.
- #define **MAX_FILENAME_LEN** 64

  Space to allocate for filename char array.
- #define **MAX_SUFFIX_LEN** 32

  Space to allocate for filename suffix.
- #define **IO_METHOD_COUNT** 7

  How many IO methods to accept.
- #define **STRIPE_TYPE_COUNT** 3

  How many striping types to accept.
- #define **SERIAL_IO_IDX** 0

  Index of the 'serial' IO method in io_method_names of benchio.c.
- #define **PROC_IO_IDX** 1

Index of the 'proc' IO method in io_method_names of benchio.c.

- #define **NODE_IO_IDX** 2

  Index of the 'node' IO method in io_method_names of benchio.c.
- #define **MPIIO_IO_IDX** 3

  Index of the 'mpiio' IO method in io_method_names of benchio.c.
- #define **ADIOS2_IO_IDX** 6

  Index of the 'adios' IO method in io_method_names of benchio.c.
- #define **KIB** 1024

  A base-2 kilobyte (a kibibyte, 2^10 bytes).
- #define **MIB** (KIB ∗ KIB)

  A base-2 megabyte (a mibibyte, 2^20 bytes).
- #define **GIB** (KIB ∗ MIB)

  A base-2 gigabyte (a gibibyte, 2^30 bytes).
- #define **NULLCHAR** '\0'

  Character which terminates strings.
- #define **LONG_ARG** "--"

  If user input starts with this, it indicates a long argument is to be provided.
- #define **SHORT_ARG** "-"

  If user input starts with this, it indicates a short argument is to be provided.
- #define **SEPARATOR_STRING** "----------\n"

  Standard line to print out. Defined to make sure all lines are same length.
- #define DEBUG_MODE false

  Whether or not to enable automatic correctness testing (slow for large files).
- #define **DEBUG_PRINT_NUM_VALUES** 5

  In debug mode, number of values to print out before a detected error.
- #define **ADIOS_CONFIG_FILE** "adios2_config.xml"

  Where to read adios configuration from.
- #define **ADIOS_GLOBAL_ARRAY_VAR** "adios_global_array"

  What to name the global array variable of adios.
- #define **ADIOS_IO_NAME** "adios_output"

  Name of the IO handler in adios2 to look for from config.
- #define **ADIOS_MODE_HDF5** "hdf5"

  ADIOS engine type string representing HDF5.
- #define **ADIOS_MODE_BP3** "bp3"

  ADIOS engine type string representing BP3.
- #define **ADIOS_MODE_BP4** "bp4"

  ADIOS engine type string representing BP4.
- #define **ADIOS_MODE_BP5** "bp5"

  ADIOS engine type string representing BP5.
- #define **ADIOS_BP3_FILE_COUNT** 3

  How many extra files generated by ADIOS' BP3 format to delete.
- #define **ADIOS_BP4_FILE_COUNT** 4

  How many extra files generated by ADIOS' BP4 format to delete.
- #define **ADIOS_BP5_FILE_COUNT** 5

  How many extra files generated by ADIOS' BP5 format to delete.

Enumerations

- enum io_mode {
  none , **serial** , **mpiio** , **adios_hdf5** ,
  adios_bp3 , **adios_bp4** , **adios_bp5** }

  Used to keep internal track of what io mode is being used.

Functions

- bool process_args (int argc, char **argv, int my_rank, int *sizes, bool *global_flag, bool *use_io_method, bool *use_stripe_method, bool *read_benchmark, bool *user_wants_help)

  Parse command-line arguments provided by user.
- void setup_nodes (MPI_Comm communicator, int original_rank, MPI_Comm *node_comm, int *node_↩ number)

  Set up communicators reflecting the nodal environment currently in use.
- void populate_io_data (double *local_sizes, int *coords, double *global_sizes, double ***io_data)

  Fill the current process' array with data consistent with the global array.
- void main_benchmark_loop (MPI_Comm cartesian_comm, MPI_Comm node_comm, bool *use_stripe↩ _method, bool *use_io_method, double *local_sizes, double *global_sizes, double global_size_gib, bool read_benchmark, int node_num, enum io_mode adios_io_mode, int *coords, double ***io_data)

  Start the benchmark, looping through each step according to configuration, and report results.
- bool run_read_benchmark (char *file_name, double *local_sizes, double *global_sizes, MPI_Comm io_↩ comm, int my_rank, int io, double global_size_gib, enum io_mode io_mode, MPI_Comm cartesian_comm, int *coords)

  Run the read-based benchmark according to specified configuration.
- bool run_write_benchmark (char *file_name, double ***io_data, double *local_sizes, double *global_sizes, MPI_Comm io_comm, int my_rank, int io, double global_size_gib, enum io_mode io_mode)

  Run the write-based benchmark according to specified configuration.
- bool equals_ignore_case (char const *string_one, char const *string_two)

  Check if two strings are the same in a case-insensitive way.
- bool string_to_integer (char *number_string, int *number_int)

  Convert a char array to an integer.
- void *** arraymalloc3d (int nx, int ny, int nz, size_t typesize)

  Allocate a continuous 3D array of specified dimensions.
- bool boss_delete (enum io_mode io_mode, char const *file_name, MPI_Comm communicator)

  Delete the specified file from rank 0 of the passed communicator.
- void verify_output (int io_mode, char *file_name, double *global_sizes, MPI_Comm io_comm)

  Read data which has been written by the benchmark back and verify its correctness.
- void verify_input (MPI_Comm cartesian_comm, double ***io_data, int *coords, double *local_sizes, double *global_sizes)

  Verify that data was read back correctly.
- void **print_simple_usage** ()

  Print a simple instruction manual to the user.
- void **print_detailed_usage** ()

  Print detailed instructions of how to use benchio to the user.
- void serial_write (char const *file_name, double ***io_data, double *local_sizes, MPI_Comm communicator)

  Perform an write to disk from rank 0 of the specified communicator.
- void serial_read (char const *file_name, double ***io_data, double *local_sizes, MPI_Comm communicator)

  Perform an read from disk on rank 0 of the specified communicator.
- void mpiio_write (char const *file_name, double ***io_data, double *local_sizes, double *global_sizes, MPI_Comm cartesian_comm)

  Perform an MPI-IO write of the global array to file.
- void mpiio_read (char const *file_name, double ***io_data, double *local_sizes, double *global_sizes, MPI_Comm cartesian_comm)

  Perform an MPI-IO read of the global array from file.
- void adios2_write (char const *file_name, double ***io_data, double *local_sizes, double *global_sizes, MPI_Comm cartesian_comm)

  Perform an ADIOS2 write of the global array to file.
- void adios2_read (char const *file_name, double ***io_data, double *local_sizes, double *global_sizes, MPI_Comm cartesian_comm)

Perform an ADIOS2 read of the global array from file.

- void adios2_verify (char *file_name, double *global_sizes, MPI_Comm communicator)

  Read data which has been written with ADIOS2 back and verify its correctness.
- bool get_adios2_io_mode (MPI_Comm io_comm, enum io_mode *adios_io_mode)

  Retreive the current ADIOS2 IO mode in use.
- bool adios2_native_cleanup (char const *file_name, enum io_mode io_mode)

  Clean up the files written by the native ADIOS modes.

5.3.1 Detailed Description

Header file for benchio program.

Contains readability constants and function headers for the whole of benchio.

5.3.2 LICENSE

This software is released under the MIT License.

5.3.3 Macro Definition Documentation

5.3.3.1 DEBUG_MODE

```
#define DEBUG_MODE false
```

Whether or not to enable automatic correctness testing (slow for large files).

Can use makefile to enable with command "DEBUG_MODE=true make"

5.3.4 Enumeration Type Documentation

5.3.4.1 io_mode

```
enum io_mode
```

Used to keep internal track of what io mode is being used.

Mostly useful for adios, which can use different underlying io modes with different behavior.

5.3.5 Function Documentation

5.3.5.1 adios2_native_cleanup()

```
bool adios2_native_cleanup (
            char const * file_name,
            enum io_mode io_mode)
```

Clean up the files written by the native ADIOS modes.

Dummy function if compiled without ADIOS2. See implementation in adios2.c for detailed documentation.

Remove the files written by ADIOS2 modes bp3, bp4, or bp5. Since ADIOS2 writes a folder, with the files of the folder varying depending on which mode is used, and the aggregator count, this removal uses 'brute force' to accomplish this; see details.

Parameters

in	*file_name*	The name of the file or resource to clean up.
in	*io_mode*	The ADIOS2 IO mode being used

Returns

Returns `true` if cleanup was successful, `false` otherwise.

This is a bit of a workaround for deletion with ADIOS, which generates a set of files for its native binary formats. The goal is to have a platform-independent solution to delete whatever file was generated. Unfortunately not many good options exist, so solution here is to delete files one-by-one depending on mode used. An additional complication is that many numbered files are sometimes generated, e.g. data.0, data.1, ... This is solved here with a do-while loop, which just keeps incrementing a counter and deleting as long as it is successful.

5.3.5.2 adios2_read()

```
void adios2_read (
            char const * file_name,
            double *** io_data,
            double * local_sizes,
            double * global_sizes,
            MPI_Comm cartesian_comm)
```

Perform an ADIOS2 read of the global array from file.

Dummy function if compiled without ADIOS2. See implementation in adios2.c for detailed documentation.

This function reads data from a file using the ADIOS2 library. It reads the data into the `io_data` array using the specified file name, local and global sizes, and MPI communicator.

Parameters

in	*file_name*	Name of the ADIOS2 file (folder) to read from
in,out	*io_data*	The pre-allocated current process' 3D array to fill with data from file
in	*local_sizes*	An array of values indicating the per-process array dimensions
in	*global_sizes*	An array of values indicating the global array dimensions
in	*cartesian_comm*	An MPI communicator with a cartesian topology

5.3.5.3 adios2_verify()

```
void adios2_verify (
            char * file_name,
            double * global_sizes,
            MPI_Comm communicator)
```

Read data which has been written with ADIOS2 back and verify its correctness.

Dummy function if compiled without ADIOS2. See implementation in adios2.c for detailed documentation.

Use a single process to read the ADIOS2 data which was written to file through the benchmark, loading it into a 1D array and verifying that it forms the correct pattern (a series of incrementing double-precision values starting from 1 and ending at global_sizes[0]∗global_sizes[1]∗global_sizes[2])

Parameters

in	*file_name*	Name of the ADIOS2 file (folder) to read from
in	*global_sizes*	An array of values indicating the global array dimensions
in	*communicator*	The global MPI communicator

5.3.5.4 adios2_write()

```
void adios2_write (
            char const * file_name,
            double *** io_data,
            double * local_sizes,
            double * global_sizes,
            MPI_Comm cartesian_comm)
```

Perform an ADIOS2 write of the global array to file.

Dummy function if compiled without ADIOS2. See implementation in adios2.c for detailed documentation.

Write the global array, stored as a set of local arrays in `io_data`, to the specified file, using ADIOS2. Perform error error checking throughout to report any issues encountered.

Parameters

in	*file_name*	Name of the ADIOS2 file (folder) to create
in	*io_data*	The current process' 3D array of data forming its part of the global array
in	*local_sizes*	An array of values indicating the per-process array dimensions
in	*global_sizes*	An array of values indicating the global array dimensions
in	*cartesian_comm*	An MPI communicator with a cartesian topology

5.3.5.5 arraymalloc3d()

```
void *** arraymalloc3d (
            int nx,
            int ny,
            int nz,
            size_t typesize)
```

Allocate a continuous 3D array of specified dimensions.

See implementation in benchutil.c for detailed documentation.

Dynamically allocate a 3D continuous array of specified type and dimensions. Important: To function correctly, pass `my_array[0][0][0]` to any libraries functions, instead of just `my_array`.

Parameters

in	*nx*	Size (number of elements) of the array to create in the first dimension
in	*ny*	Size (number of elements) of the array to create in the second dimension
in	*nz*	Size (number of elements) of the array to create in the third dimension
in	*typesize*	Size (bytes) of each element in the array

Returns

Pointer to 3D allocated data

Code provided by David Henty, from: https://github.com/davidhenty/bcastc/

Author

David Henty

5.3.5.6 boss_delete()

```
bool boss_delete (
            enum io_mode io_mode,
            char const * file_name,
            MPI_Comm communicator)
```

Delete the specified file from rank 0 of the passed communicator.

See implementation in benchutil.c for detailed documentation.

Parameters

in	*io_mode*	The IO mode that was used for the write
in	*file_name*	The name of the file/folder to delete
in	*communicator*	The global MPI communicator

Returns

Returns `true` if the file was succesfully deleted, otherwise `false`

5.3.5.7 equals_ignore_case()

```
bool equals_ignore_case (
            char const * string_one,
            char const * string_two)
```

Check if two strings are the same in a case-insensitive way.

See implementation in benchutil.c for detailed documentation.

Compare two strings character by character to see if they are the same, but convert each character to lowercase before making the comparison.

Parameters

in	*string_one*	First character array (string)
in	*string_two*	Second character array (string)

Returns

Return `true` if the two strings were the same (ignoring case) and `false` otherwise.

5.3.5.8 get_adios2_io_mode()

```
bool get_adios2_io_mode (
            MPI_Comm io_comm,
            enum io_mode * adios_io_mode)
```

Retreive the current ADIOS2 IO mode in use.

Dummy function if compiled without ADIOS2 with a warning printout if called. See implementation in adios2.c for detailed documentation.

Starts ADIOS2 from the configuration file and reads the IO engine to use

Parameters

in	*io_comm*	The global MPI communicator
out	*adios_io_mode*	Pointer to enum io_mode where to store the retreived IO mode

Returns

Returns `true` if successfully read the IO mode of ADIOS2, `false` otherwise.

5.3.5.9 main_benchmark_loop()

```
void main_benchmark_loop (
            MPI_Comm cartesian_comm,
            MPI_Comm node_comm,
            bool * use_stripe_method,
            bool * use_io_method,
            double * local_sizes,
            double * global_sizes,
            double global_size_gib,
            bool read_benchmark,
            int node_num,
            enum io_mode adios_io_mode,
            int * coords,
            double *** io_data)
```

Start the benchmark, looping through each step according to configuration, and report results.

See implementation in benchio.c for detailed documentation.

Take some identified set of settings, such as the IO methods to use, the stripings to use, and whether or not to run a read-back benchmark, and run the benchmark accordingly.

Parameters

in	*cartesian_comm*	An MPI communicator with a cartesian topology
in	*node_comm*	An MPI communicator specific to the current process' node
in	*use_stripe_method*	An array of booleans indicating whether or not to use each striping method
in	*use_io_method*	An array of booleans indicating whether or not to use each io method
in	*local_sizes*	An array of values indicating the per-process array dimensions
in	*global_sizes*	An array of values indicating the global array dimensions

in	*global_size_gib*	The total global array size in GiB.
in	*read_benchmark*	Boolean flag indicating whether or not to also perform a read benchmark
in	*node_num*	The rank of the process' current node in the node-spanning communicator
in	*adios_io_mode*	Which underlying IO mode ADIOS is using, if applicable
in	*coords*	The coordinates of the current process in cartesian_comm
in	*io_data*	The current process' 3D array of data forming its part of the global array

5.3.5.10 mpiio_read()

```
void mpiio_read (
            char const * file_name,
            double *** io_data,
            double * local_sizes,
            double * global_sizes,
            MPI_Comm cartesian_comm)
```

Perform an MPI-IO read of the global array from file.

See implementation in mpiio.c for detailed documentation.

This function reads data from a file using the MPI-IO library. It reads the data into the `io_data` array using the specified file name, local and global sizes, and MPI communicator.

Parameters

in	*file_name*	Name of the (MPI-IO / binary) file to read from
in, out	*io_data*	The pre-allocated current process' 3D array to fill with data from file
in	*local_sizes*	An array of values indicating the per-process array dimensions
in	*global_sizes*	An array of values indicating the global array dimensions
in	*cartesian_comm*	An MPI communicator with a cartesian topology

5.3.5.11 mpiio_write()

```
void mpiio_write (
            char const * file_name,
            double *** io_data,
            double * local_sizes,
            double * global_sizes,
            MPI_Comm cartesian_comm)
```

Perform an MPI-IO write of the global array to file.

See implementation in mpiio.c for detailed documentation.

Write the global array, stored as a set of local arrays in `io_data`, to the specified file, using MPI-IO. Perform error error checking throughout to report any issues encountered.

Parameters

in	*file_name*	Name of the MPI-IO file to create
in	*io_data*	The current process' 3D array of data forming its part of the global array
in	*local_sizes*	An array of values indicating the per-process array dimensions
in	*global_sizes*	An array of values indicating the global array dimensions
in	*cartesian_comm*	An MPI communicator with a cartesian topology

5.3.5.12 populate_io_data()

```
void populate_io_data (
            double * local_sizes,
            int * coords,
            double * global_sizes,
            double *** io_data)
```

Fill the current process' array with data consistent with the global array.

See implementation in benchio.c for detailed documentation.

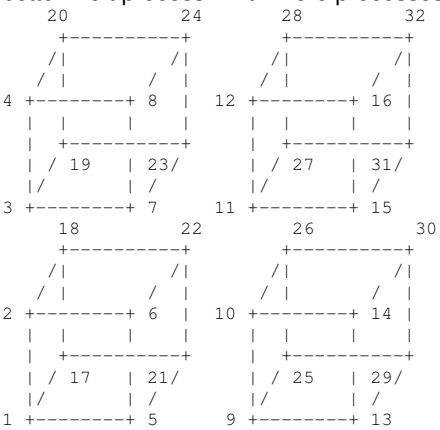
Fill the current process' array with data such that, in the cartesian grid of processes, it forms part of a series of incrementing double-precision values from 1 to global_sizes[0]∗global_sizes[1]∗global_sizes[2].

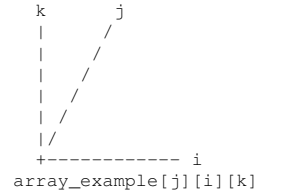
Parameters

in	*local_sizes*	The array of sizes on the current process
in	*coords*	The array of coordinates in the cartesian communuicator
in	*global_sizes*	The array of sizes of the global array
in,out	*io_data*	Pointer to a pre-allocated 3D array to populate

Populate the array of the current process. Each element contains a globally unique double-precision value between 1 and global_sizes[0]∗global_sizes[1]∗global_sizes[2]. Count increments by one for each element.

Example ordering on a (2, 2, 1) decomposition with 2x2x2 local data. Each cube is a process' own array. P0 is bottom-left process. With more processes, extend pattern...






```
array_example[j][i][k]

Notice that MPI structures its process grid as follows:
mpi_dims_example[i][k][j]
```

5.3.5.13 process_args()

```
bool process_args (
            int argc,
            char ** argv,
            int my_rank,
            int * sizes,
            bool * global_flag,
            bool * use_io_method,
            bool * use_stripe_method,
            bool * read_benchmark,
            bool * user_wants_help)
```

Parse command-line arguments provided by user.

See implementation in benchio.c for detailed documentation.

The the command-line arguments provided by the user and validate them. Print useful debug to user and return if some of their input was invalid, otherwise quietly fill the parameters with the specified input.

Parameters

in	*argc*	Number of arguments from user
in	*argv*	Argument array from user
in	*my_rank*	Current process rank in the global MPI communicator
out	*sizes*	Array of size DIMENSIONS to store the user input for dimensions
out	*global_flag*	Pointer to boolean to store flag of whether to use local or global scaling
out	*use_io_method*	Array of size IO_METHOD_COUNT to store boolean values indicating whether to use each IO method
out	*use_stripe_method*	Array of size STRIPE_TYPE_COUNT to store boolean values indicating whether to use each striping method
out	*read_benchmark*	Pointer to a boolean to store flag of whether to also perform read-based benchmark
out	*user_wants_help*	Pointer to a boolean to store flag of whether user asked for help through –help or -h flags

Returns

Return `true` if OK to continue benchmark, otherwise `false`

5.3.5.14 run_read_benchmark()

```
bool run_read_benchmark (
            char * file_name,
            double * local_sizes,
            double * global_sizes,
            MPI_Comm io_comm,
            int my_rank,
            int io,
            double global_size_gib,
            enum io_mode io_mode,
            MPI_Comm cartesian_comm,
            int * coords)
```

Run the read-based benchmark according to specified configuration.

See implementation in benchio.c for detailed documentation.

Run the read-based benchmark and report results. If DEBUG_MODE is enabled, this function also validates the data which was read back in for correctness.

Parameters

in	*file_name*	Name of the file to read from
in	*local_sizes*	An array of values indicating the per-process array dimensions
in	*global_sizes*	An array of values indicating the global array dimensions
in	*io_comm*	The MPI communicator context to perform the read in
in	*my_rank*	The rank of the current process in the global communicator
in	*io*	An integer indicating which IO mode to read using; see benchio.h compile-time constants
in	*global_size_gib*	The size of the global array in GiB.
in	*io_mode*	Which underlying IO mode is in use (ADIOS2 has multiple engines)
in	*cartesian_comm*	The cartesian-topology communicator which was used to populate data
in	*coords*	Coordinates of current process in cartesian_comm, size DIMENSIONS

Returns

Returns `true` if benchmark completed normally, `false` otherwise.

5.3.5.15 run_write_benchmark()

```
bool run_write_benchmark (
            char * file_name,
            double *** io_data,
            double * local_sizes,
            double * global_sizes,
            MPI_Comm io_comm,
            int my_rank,
            int io,
            double global_size_gib,
            enum io_mode io_mode)
```

Run the write-based benchmark according to specified configuration.

Run the write-based benchmark and report results. If DEBUG_MODE is enabled, this function also validates the data which was written by reading it back in and checking it for correctness.

Parameters

in	*file_name*	Name of the file to read from
in	*io_data*	The current process' 3D array of data forming its part of the global array
in	*local_sizes*	An array of values indicating the per-process array dimensions
in	*global_sizes*	An array of values indicating the global array dimensions
in	*io_comm*	The MPI communicator context to perform the write in
in	*my_rank*	The rank of the current process in the global communicator
in	*io*	An integer indicating which IO mode to read using; see benchio.h compile-time constants
in	*global_size_gib*	The size of the global array in GiB.
in	*io_mode*	Which underlying IO mode is in use (ADIOS2 has multiple engines)

Returns

Returns `true` if benchmark completed normally, `false` otherwise.

5.3.5.16 serial_read()

```
void serial_read (
            char const * file_name,
            double *** io_data,
            double * local_sizes,
            MPI_Comm communicator)
```

Perform an read from disk on rank 0 of the specified communicator.

See implementation in serial.c for detailed documentation.

Read data back in the same way it was written using serial_write.

Parameters

in	*file_name*	Name of the file to open to read
in	*io_data*	3D array to fill with data from the read
in	*local_sizes*	An array of values indicating the per-process array dimensions
in	*communicator*	Communicator from which to check if rank is 0 and should perform read

5.3.5.17 serial_write()

```
void serial_write (
            char const * file_name,
            double *** io_data,
            double * local_sizes,
            MPI_Comm communicator)
```

Perform an write to disk from rank 0 of the specified communicator.

See implementation in serial.c for detailed documentation.

Write same amount of data as the parallel write but do it all from rank 0 This is just to get a baseline figure for serial IO performance - note that the contents of the file will be different from the parallel calls.

Parameters

in	*file_name*	Name of the file to create
in	*io_data*	The current process' 3D array of data forming its part of the global array
in	*local_sizes*	An array of values indicating the per-process array dimensions
in	*communicator*	Communicator from which to check if rank is 0 and should perform write

5.3.5.18 setup_nodes()

```
void setup_nodes (
            MPI_Comm communicator,
            int original_rank,
            MPI_Comm * node_comm,
            int * node_number)
```

Set up communicators reflecting the nodal environment currently in use.

See implementation in benchio.c for detailed documentation.

Create node-spanning communicators and identify the node bosses (rank 0s) of each node. Also identify and print out the name of each node.

Parameters

in	*communicator*	An MPI communicator with all processes in it
in	*original_rank*	Rank of current process within `communicator`
out	*node_comm*	Pointer to communicator to fill with a communicator spanning the current node
out	*node_number*	Pointer to integer which will be filled with the node number of the current process

5.3.5.19 string_to_integer()

```
bool string_to_integer (
            char * number_string,
            int * number_int)
```

Convert a char array to an integer.

See implementation in benchutil.c for detailed documentation.

Take some char array string number_string and convert it to an integer assuming checks pass. The checks confirm that the user input only contains digits, is over 0, and is within the integer bounds.

Parameters

in	*number_string*	The number, as a character array
out	*number_int*	The number as an integer if applicable

Returns

Return `true` if number successfully parsed, otherwise `false`

5.3.5.20 verify_input()

```
void verify_input (
            MPI_Comm cartesian_comm,
            double *** io_data,
            int * coords,
            double * local_sizes,
            double * global_sizes)
```

Verify that data was read back correctly.

See implementation in benchutil.c for detailed documentation.

For the read-based benchmark, verify that each process holds the expected data. Used in debug mode.

Parameters

in	*cartesian_comm*	An MPI communicator with a cartesian topology
in	*io_data*	The current process' 3D array of data forming its part of the global array
in	*coords*	The coordinates of the current process in cartesian_comm
in	*global_sizes*	An array of values indicating the local array dimensions
in	*global_sizes*	An array of values indicating the global array dimensions

5.3.5.21 verify_output()

```
void verify_output (
            int io_mode,
            char * file_name,
            double * global_sizes,
            MPI_Comm io_comm)
```

Read data which has been written by the benchmark back and verify its correctness.

See implementation in benchutil.c for detailed documentation.

Use a single process to read the data which was written to file through the benchmark from file, number by number, checking that it forms the correct pattern (a series of incrementing double-precision values starting from 1 and ending at global_sizes[0]∗global_sizes[1]∗global_sizes[2]). If ADIOS2 was used, call another function to verify since it works differently.

Parameters

in	*io_mode*	The IO mode that was used for the write
in	*file_name*	The name of the file/folder to verify correctness of
in	*global_sizes*	An array of values indicating the global array dimensions
in	*io_comm*	The global MPI communicator

5.4 benchio.h

Go to the documentation of this file.

```
00001
00012 #ifndef BENCHIO_H
00013 #define BENCHIO_H
00014
00015 #include <stdbool.h>
00016 #include <stdio.h>
00017 #include <stdlib.h>
00018 #include <mpi.h>
00019
00020 /************************* Readability Constants **********************/
00021
00026 #define RANK_ZERO 0
00027
00032 #define DIMENSIONS 3
00033
00038 #define MAX_FILENAME_LEN 64
00039
00044 #define MAX_SUFFIX_LEN 32
00045
00050 #define IO_METHOD_COUNT 7
00051
00056 #define STRIPE_TYPE_COUNT 3
00057
00062 #define SERIAL_IO_IDX 0
00063
00068 #define PROC_IO_IDX 1
00069
00074 #define NODE_IO_IDX 2
00075
00080 #define MPIIO_IO_IDX 3
00081
00086 #define ADIOS2_IO_IDX 6
00087
00092 #define KIB 1024
00093
00098 #define MIB (KIB * KIB)
00099
00104 #define GIB (KIB * MIB)
00105
00110 #define NULLCHAR '\0'
```

```
00111
00116 #define LONG_ARG "--"
00117
00122 #define SHORT_ARG "-"
00123
00128 #define SEPARATOR_STRING "----------------------------\n"
00129
00130 /******************************** Debug Mode ***************************/
00131
00132 #ifndef DEBUG_MODE
00133
00140 #define DEBUG_MODE false
00141
00142 #endif
00143
00148 #define DEBUG_PRINT_NUM_VALUES 5
00149
00150 /******************************** Enums ********************************/
00151
00155 struct argument
00156 {
00157     char *long_arg;
00158     char *short_arg;
00159     bool standalone;
00160     bool optional;
00161     bool found;
00162     bool complete;
00163 };
00164
00170 enum io_mode
00171 {
00172     none,
00173     serial,
00174     mpiio,
00175     adios_hdf5,
00176     adios_bp3,
00177     adios_bp4,
00178     adios_bp5
00179 };
00180
00181 /******************************** benchio.c ****************************/
00182
00188 bool process_args(int argc, char **argv, int my_rank, int *sizes, bool *global_flag, bool
      *use_io_method, bool *use_stripe_method, bool *read_benchmark, bool *user_wants_help);
00189
00195 void setup_nodes(MPI_Comm communicator, int original_rank, MPI_Comm *node_comm, int *node_number);
00196
00202 void populate_io_data(double *local_sizes, int *coords, double *global_sizes, double ***io_data);
00203
00209 void main_benchmark_loop(MPI_Comm cartesian_comm, MPI_Comm node_comm, bool *use_stripe_method, bool
      *use_io_method, double *local_sizes,
00210                          double *global_sizes, double global_size_gib, bool read_benchmark, int
      node_num, enum io_mode adios_io_mode,
00211                          int *coords, double ***io_data);
00217 bool run_read_benchmark(char *file_name, double *local_sizes, double *global_sizes, MPI_Comm io_comm,
      int my_rank, int io, double global_size_gib, enum io_mode io_mode, MPI_Comm cartesian_comm, int
      *coords);
00218
00222 bool run_write_benchmark(char *file_name, double ***io_data, double *local_sizes, double
      *global_sizes, MPI_Comm io_comm, int my_rank, int io, double global_size_gib, enum io_mode io_mode);
00223
00224 /******************************** benchutil.c **************************/
00225
00231 bool equals_ignore_case(char const *string_one, char const *string_two);
00232
00238 bool string_to_integer(char *number_string, int *number_int);
00239
00245 void ***arraymalloc3d(int nx, int ny, int nz, size_t typesize);
00246
00252 bool boss_delete(enum io_mode io_mode, char const *file_name, MPI_Comm communicator);
00253
00259 void verify_output(int io_mode, char *file_name, double *global_sizes, MPI_Comm io_comm);
00260
00266 void verify_input(MPI_Comm cartesian_comm, double ***io_data, int *coords, double *local_sizes, double
      *global_sizes);
00267
00271 void print_simple_usage();
00272
00276 void print_detailed_usage();
00277
00278 /******************************** serial.c *****************************/
00279
00285 void serial_write(char const *file_name, double ***io_data, double *local_sizes, MPI_Comm
      communicator);
00286
00292 void serial_read(char const *file_name, double ***io_data, double *local_sizes, MPI_Comm
      communicator);
```

```
00293
00294 /****************************** mpiio.c ******************************/
00295
00301 void mpiio_write(char const *file_name, double ***io_data, double *local_sizes, double *global_sizes,
      MPI_Comm cartesian_comm);
00302
00308 void mpiio_read(char const *file_name, double ***io_data, double *local_sizes, double *global_sizes,
      MPI_Comm cartesian_comm);
00309
00310 /****************************** adios2.c ******************************/
00311
00316 #define ADIOS_CONFIG_FILE "adios2_config.xml"
00317
00322 #define ADIOS_GLOBAL_ARRAY_VAR "adios_global_array"
00323
00328 #define ADIOS_IO_NAME "adios_output"
00329
00334 #define ADIOS_MODE_HDF5 "hdf5"
00335
00340 #define ADIOS_MODE_BP3 "bp3"
00341
00346 #define ADIOS_MODE_BP4 "bp4"
00347
00352 #define ADIOS_MODE_BP5 "bp5"
00353
00358 #define ADIOS_BP3_FILE_COUNT 3
00359
00364 #define ADIOS_BP4_FILE_COUNT 4
00365
00370 #define ADIOS_BP5_FILE_COUNT 5
00371
00378 void adios2_write(char const *file_name, double ***io_data, double *local_sizes, double *global_sizes,
      MPI_Comm cartesian_comm);
00379
00386 void adios2_read(char const *file_name, double ***io_data, double *local_sizes, double *global_sizes,
      MPI_Comm cartesian_comm);
00387
00394 void adios2_verify(char *file_name, double *global_sizes, MPI_Comm communicator);
00395
00402 bool get_adios2_io_mode(MPI_Comm io_comm, enum io_mode *adios_io_mode);
00403
00410 bool adios2_native_cleanup(char const *file_name, enum io_mode io_mode);
00411
00412 #endif
```

5.5 benchutil.c File Reference

Utility functions for benchio program.

```
#include <ctype.h>
#include <unistd.h>
#include <errno.h>
#include <limits.h>
#include "benchio.h"
```

Functions

- void verify_output (int io_mode, char *file_name, double *global_sizes, MPI_Comm io_comm)

  Read data which has been written by the benchmark back and verify its correctness.
- void verify_input (MPI_Comm cartesian_comm, double ***io_data, int *coords, double *local_sizes, double *global_sizes)

  Verify that data was read back correctly.
- bool equals_ignore_case (char const *string_one, char const *string_two)

  Check if two strings are the same in a case-insensitive way.
- bool string_to_integer (char *number_string, int *number_int)

  Convert a char array to an integer.
- bool boss_delete (enum io_mode io_mode, char const *file_name, MPI_Comm communicator)

Delete the specified file from rank 0 of the passed communicator.

- void *** arraymalloc3d (int nx, int ny, int nz, size_t typesize)

  Allocate a continuous 3D array of specified dimensions.
- void **print_simple_usage** ()

  Print a simple instruction manual to the user.
- void **print_detailed_usage** ()

  Print detailed instructions of how to use benchio to the user.

5.5.1 Detailed Description

Utility functions for benchio program.

Contains useful utility functions for benchio, including some functions used for debug mode.

5.5.2 LICENSE

This software is released under the MIT License.

5.5.3 Function Documentation

5.5.3.1 arraymalloc3d()

```
void *** arraymalloc3d (
            int nx,
            int ny,
            int nz,
            size_t typesize)
```

Allocate a continuous 3D array of specified dimensions.

Dynamically allocate a 3D continuous array of specified type and dimensions. Important: To function correctly, pass `my_array[0][0][0]` to any libraries functions, instead of just `my_array`.

Parameters

in	*nx*	Size (number of elements) of the array to create in the first dimension
in	*ny*	Size (number of elements) of the array to create in the second dimension
in	*nz*	Size (number of elements) of the array to create in the third dimension
in	*typesize*	Size (bytes) of each element in the array

Returns

Pointer to 3D allocated data

Code provided by David Henty, from: https://github.com/davidhenty/bcastc/

Author

David Henty

5.5.3.2 boss_delete()

```
bool boss_delete (
            enum io_mode io_mode io_mode ,
            char const * file_name,
            MPI_Comm communicator)
```

Delete the specified file from rank 0 of the passed communicator.

Parameters

in	*io_mode*	The IO mode that was used for the write
in	*file_name*	The name of the file/folder to delete
in	*communicator*	The global MPI communicator

Returns

Returns `true` if the file was succesfully deleted, otherwise `false`

5.5.3.3 equals_ignore_case()

```
bool equals_ignore_case (
            char const * string_one,
            char const * string_two)
```

Check if two strings are the same in a case-insensitive way.

Compare two strings character by character to see if they are the same, but convert each character to lowercase before making the comparison.

Parameters

in	*string_one*	First character array (string)
in	*string_two*	Second character array (string)

Returns

Return `true` if the two strings were the same (ignoring case) and `false` otherwise.

5.5.3.4 string_to_integer()

```
bool string_to_integer (
            char * number_string,
            int * number_int)
```

Convert a char array to an integer.

Take some char array string number_string and convert it to an integer assuming checks pass. The checks confirm that the user input only contains digits, is over 0, and is within the integer bounds.

Parameters

in	*number_string*	The number, as a character array
out	*number_int*	The number as an integer if applicable

Returns

Return `true` if number successfully parsed, otherwise `false`

5.5.3.5 verify_input()

```
void verify_input (
            MPI_Comm cartesian_comm,
            double *** io_data,
            int * coords,
            double * local_sizes,
            double * global_sizes)
```

Verify that data was read back correctly.

For the read-based benchmark, verify that each process holds the expected data. Used in debug mode.

Parameters

in	*cartesian_comm*	An MPI communicator with a cartesian topology
in	*io_data*	The current process' 3D array of data forming its part of the global array
in	*coords*	The coordinates of the current process in cartesian_comm
in	*global_sizes*	An array of values indicating the local array dimensions
in	*global_sizes*	An array of values indicating the global array dimensions

5.5.3.6 verify_output()

```
void verify_output (
            int io_mode,
            char * file_name,
            double * global_sizes,
            MPI_Comm io_comm)
```

Read data which has been written by the benchmark back and verify its correctness.

Use a single process to read the data which was written to file through the benchmark from file, number by number, checking that it forms the correct pattern (a series of incrementing double-precision values starting from 1 and ending at global_sizes[0]∗global_sizes[1]∗global_sizes[2]). If ADIOS2 was used, call another function to verify since it works differently.

Parameters

in	*io_mode*	The IO mode that was used for the write
in	*file_name*	The name of the file/folder to verify correctness of
in	*global_sizes*	An array of values indicating the global array dimensions
in	*io_comm*	The global MPI communicator

5.6 mpiio.c File Reference

File for MPI-IO related code in benchio.

```
#include "benchio.h"
```

Functions

- void mpiio_write (char const *file_name, double ***io_data, double *local_sizes, double *global_sizes, MPI_Comm cartesian_comm)

  Perform an MPI-IO write of the global array to file.
- void mpiio_read (char const *file_name, double ***io_data, double *local_sizes, double *global_sizes, MPI_Comm cartesian_comm)

  Perform an MPI-IO read of the global array from file.

5.6.1 Detailed Description

File for MPI-IO related code in benchio.

Contains code relating to reading and writing MPI-IO files.

5.6.2 LICENSE

This software is released under the MIT License.

5.6.3 Function Documentation

5.6.3.1 mpiio_read()

```
void mpiio_read (
            char const * file_name,
            double *** io_data,
            double * local_sizes,
            double * global_sizes,
            MPI_Comm cartesian_comm)
```

Perform an MPI-IO read of the global array from file.

This function reads data from a file using the MPI-IO library. It reads the data into the `io_data` array using the specified file name, local and global sizes, and MPI communicator.

Parameters

`in`	*file_name*	Name of the (MPI-IO / binary) file to read from
`in,out`	*io_data*	The pre-allocated current process' 3D array to fill with data from file
`in`	*local_sizes*	An array of values indicating the per-process array dimensions
`in`	*global_sizes*	An array of values indicating the global array dimensions
`in`	*cartesian_comm*	An MPI communicator with a cartesian topology

5.6.3.2 mpiio_write()

```
void mpiio_write (
            char const * file_name,
            double *** io_data,
            double * local_sizes,
            double * global_sizes,
            MPI_Comm cartesian_comm)
```

Perform an MPI-IO write of the global array to file.

Write the global array, stored as a set of local arrays in `io_data`, to the specified file, using MPI-IO. Perform error error checking throughout to report any issues encountered.

Parameters

in	*file_name*	Name of the MPI-IO file to create
in	*io_data*	The current process' 3D array of data forming its part of the global array
in	*local_sizes*	An array of values indicating the per-process array dimensions
in	*global_sizes*	An array of values indicating the global array dimensions
in	*cartesian_comm*	An MPI communicator with a cartesian topology

5.7 serial.c File Reference

File for serial/node/proc IO modes of benchio.

```
#include "benchio.h"
```

Functions

- void serial_write (char const *file_name, double ***io_data, double *local_sizes, MPI_Comm communicator)
  Perform an write to disk from rank 0 of the specified communicator.
- void serial_read (char const *file_name, double ***io_data, double *local_sizes, MPI_Comm communicator)
  Perform an read from disk on rank 0 of the specified communicator.

5.7.1 Detailed Description

File for serial/node/proc IO modes of benchio.

Contains code relating to reading and writing using standard IO calls.

5.7.2 LICENSE

This software is released under the MIT License.

The "serial" IO routine takes a communicator argument. This enables it to be used for a variety of purposes: MPI↩
_COMM_WORLD: standard "master" IO from a single process MPI_COMM_NODE: file-per-node MPI_COMM_↩
SELF: file-per-process

5.7.3 Function Documentation

5.7.3.1 serial_read()

```
void serial_read (
            char const * file_name,
            double *** io_data,
            double * local_sizes,
            MPI_Comm communicator)
```

Perform an read from disk on rank 0 of the specified communicator.

Read data back in the same way it was written using serial_write.

Parameters

in	*file_name*	Name of the file to open to read
in	*io_data*	3D array to fill with data from the read
in	*local_sizes*	An array of values indicating the per-process array dimensions
in	*communicator*	Communicator from which to check if rank is 0 and should perform read

5.7.3.2 serial_write()

```
void serial_write (
            char const * file_name,
            double *** io_data,
            double * local_sizes,
            MPI_Comm communicator)
```

Perform an write to disk from rank 0 of the specified communicator.

Write same amount of data as the parallel write but do it all from rank 0 This is just to get a baseline figure for serial IO performance - note that the contents of the file will be different from the parallel calls.

Parameters

in	*file_name*	Name of the file to create
in	*io_data*	The current process' 3D array of data forming its part of the global array
in	*local_sizes*	An array of values indicating the per-process array dimensions
in	*communicator*	Communicator from which to check if rank is 0 and should perform write

Index