PARALLELIZATION OF PERFORMANCE LIMITING ROUTINES IN THE
COMPUTATIONAL FLUID DYNAMICS GENERAL
NOTATION SYSTEM LIBRARY

by

Kyle Horne

A thesis submitted in partial fulfillment
of the requirements for the degree
of
MASTER OF SCIENCE
in
Mechanical Engineering

Approved:

Dr. Thomas Hauser
Major Professor

Dr. Robert Spall
Committee Member

Dr. Heng Ban
Committee Member

Dr. Byron Burnham
Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2009
Abstract

Parallelization of Performance Limiting Routines in the Computational Fluid Dynamics General Notation System Library

by

Kyle Horne, Master of Science
Utah State University, 2009

Major Professor: Dr. Thomas Hauser
Department: Mechanical and Aerospace Engineering

The Computational Fluid Dynamics General Notation System provides a unified way in which computational fluid dynamics data can be stored, but does not support the parallel I/O capabilities now available from version five of the Hierarchical Data Format library which serves as a back end for the standard. To resolve this deficiency, a new parallel extension library has been written and benchmarked for this work which can write files compliant with the standard using parallel file access modes. When using this new library, the write performance shows an increase of four-fold in some cases when compared to the same hardware operating in serial. Additionally, the use of parallel I/O allows much larger cases to be written since the problem is scattered across many nodes of a cluster, whose aggregate memory is much greater than that found on a single machine. These developments will allow computational fluid dynamics simulations to execute faster, since less time will be spent waiting for each time step to finish writing, as well prevent the need for lengthy reconstruction of data after the completion of a simulation.
Acknowledgments

Firstly, I would like to thank my wife, Lydia, for supporting me in the completion of this project. Her efforts to make our humble apartment our home have made all the difference.

Secondly, I want to thank my parents for encouraging my curiosity and scientific interest from an early age, and giving me the freedom to become who I am.

I would also like to thank Dr. Thomas Hauser for employing me since my senior year of undergraduate studies. He has funded my research and provided me with opportunities to grow that I had not even considered before. Additionally, I would thank the faculty for their efforts in making the undergraduate and graduate programs at Utah State University environments in which students can succeed. Much credit must also go to Bonnie Ogden for making sure that graduate students get all their paperwork done.

Additionally, I would thank Mr. Mike Kennedy, Mrs. Vanessa Liveris, Ms. Nicole Burt, and Mr. Gussman from Neuqua Valley High School for the sound foundation in science and mathematics which I received there.

Kyle Horne
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Scientific Computing</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Data Storage History</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2 System Level I/O</td>
<td>2</td>
</tr>
<tr>
<td>1.1.3 Hierarchical Data Format</td>
<td>3</td>
</tr>
<tr>
<td>1.1.4 Supercomputing</td>
<td>3</td>
</tr>
<tr>
<td>1.1.5 Vector Computers</td>
<td>3</td>
</tr>
<tr>
<td>1.1.6 Cluster Computing</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Computational Fluid Dynamics</td>
<td>5</td>
</tr>
<tr>
<td>1.2.1 Problem Formulation</td>
<td>5</td>
</tr>
<tr>
<td>1.2.2 Solution Algorithms</td>
<td>7</td>
</tr>
<tr>
<td>1.2.3 Parallel Implementations</td>
<td>8</td>
</tr>
<tr>
<td>1.2.4 Storage Concerns</td>
<td>8</td>
</tr>
<tr>
<td>2 Problem Description</td>
<td>13</td>
</tr>
<tr>
<td>2.1 Problem Introduction</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Problem Background</td>
<td>16</td>
</tr>
<tr>
<td>2.3 Computational Fluid Dynamics General Notation System</td>
<td>18</td>
</tr>
<tr>
<td>2.3.1 Standard Interface Data Structures</td>
<td>18</td>
</tr>
<tr>
<td>2.3.2 Advanced Data Format</td>
<td>18</td>
</tr>
<tr>
<td>2.3.3 Hierarchical Data Format version 5</td>
<td>20</td>
</tr>
</tbody>
</table>
3 Literature Review ................................................. 21
   3.1 Previous Work ................................................. 21
   3.2 Parallel Input/Output ........................................... 22
      3.2.1 Message Passing Interface Standard ......................... 22
      3.2.2 Network Attached Storage .................................. 24
      3.2.3 Message Passing Interface-I/O ............................. 24
      3.2.4 Parallel File Systems ....................................... 24
      3.2.5 Hierarchical Data Format version 5 ......................... 27

4 Parallel Implementation for the CFD General Notation System ...................... 28
   4.1 Design ......................................................... 28
   4.2 Features ....................................................... 29

5 Results .......................................................... 31
   5.1 Hardware Setup ................................................ 31
   5.2 Benchmark Selection .......................................... 33
      5.2.1 IOR ....................................................... 33
      5.2.2 pCGNS ..................................................... 33
   5.3 Benchmark Results .............................................. 34
      5.3.1 IOR ....................................................... 35
      5.3.2 pCGNS ..................................................... 38

6 Summary and Conclusion .................................................. 48

Bibliography .......................................................... 51

Appendix .............................................................. 58

A pCGNS Source Code & Documentation ........................................ 59
   A.1 Data Structure Index ............................................ 59
      A.1.1 Data Structures ............................................ 59
   A.2 File Index ........................................................ 59
      A.2.1 File List .................................................... 59
   A.3 Data Structure Documentation .................................... 60
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.3.1</td>
<td>base_s Struct Reference</td>
<td>60</td>
</tr>
<tr>
<td>A.3.2</td>
<td>coords_s Struct Reference</td>
<td>63</td>
</tr>
<tr>
<td>A.3.3</td>
<td>file_s Struct Reference</td>
<td>64</td>
</tr>
<tr>
<td>A.3.4</td>
<td>iter_s Struct Reference</td>
<td>69</td>
</tr>
<tr>
<td>A.3.5</td>
<td>section_s Struct Reference</td>
<td>70</td>
</tr>
<tr>
<td>A.3.6</td>
<td>slice_s Struct Reference</td>
<td>72</td>
</tr>
<tr>
<td>A.3.7</td>
<td>sol_s Struct Reference</td>
<td>74</td>
</tr>
<tr>
<td>A.3.8</td>
<td>zone_s Struct Reference</td>
<td>76</td>
</tr>
<tr>
<td>A.4</td>
<td>File Documentation</td>
<td>80</td>
</tr>
<tr>
<td>A.4.1</td>
<td>benchmark.c File Reference</td>
<td>80</td>
</tr>
<tr>
<td>A.4.2</td>
<td>benchmark.c</td>
<td>88</td>
</tr>
<tr>
<td>A.4.3</td>
<td>open_close.c File Reference</td>
<td>94</td>
</tr>
<tr>
<td>A.4.4</td>
<td>open_close.c</td>
<td>96</td>
</tr>
<tr>
<td>A.4.5</td>
<td>pcgns_util.c File Reference</td>
<td>97</td>
</tr>
<tr>
<td>A.4.6</td>
<td>pcgns_util.c</td>
<td>119</td>
</tr>
<tr>
<td>A.4.7</td>
<td>pcgns_util.h File Reference</td>
<td>140</td>
</tr>
<tr>
<td>A.4.8</td>
<td>pcgns_util.h</td>
<td>163</td>
</tr>
<tr>
<td>A.4.9</td>
<td>pcgnslib.c File Reference</td>
<td>168</td>
</tr>
<tr>
<td>A.4.10</td>
<td>pcgnslib.c</td>
<td>186</td>
</tr>
<tr>
<td>A.4.11</td>
<td>pcgnslib.h File Reference</td>
<td>228</td>
</tr>
<tr>
<td>A.4.12</td>
<td>pcgnslib.h</td>
<td>249</td>
</tr>
<tr>
<td>A.4.13</td>
<td>test_base.c File Reference</td>
<td>252</td>
</tr>
<tr>
<td>A.4.14</td>
<td>test_base.c</td>
<td>254</td>
</tr>
<tr>
<td>A.4.15</td>
<td>test_queue.c File Reference</td>
<td>256</td>
</tr>
<tr>
<td>A.4.16</td>
<td>test_queue.c</td>
<td>258</td>
</tr>
<tr>
<td>A.4.17</td>
<td>test_unstructured.c File Reference</td>
<td>260</td>
</tr>
<tr>
<td>A.4.18</td>
<td>test_unstructured.c</td>
<td>262</td>
</tr>
<tr>
<td>A.4.19</td>
<td>test_zone.c File Reference</td>
<td>265</td>
</tr>
<tr>
<td>A.4.20</td>
<td>test_zone.c</td>
<td>266</td>
</tr>
<tr>
<td>A.4.21</td>
<td>thesis_benchmark.c File Reference</td>
<td>268</td>
</tr>
<tr>
<td>A.4.22</td>
<td>thesis_benchmark.c</td>
<td>274</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 API of the pCGNS library as implemented in <em>pcgnslib.c</em> and declared in <em>pcgnslib.h</em>. Software using the library to access CGNS files in parallel must do so using the routines listed here.</td>
<td>30</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A generalized diagram of a cluster computer (Upper) and a shared-memory computer (Lower). The abbreviations used in the diagram are as follows: CPU=Central Processing Unit; NIC=Network Interface Card; HD=Hard Disk. The dashed line connecting all the components of each computer represents the primary data bus of the machine. The simplicity of the shared-memory computer architecture compared to that of the cluster is apparent, making shared-memory computers desirable when possible. Unfortunately, the performance required from the main data bus in a shared-memory computer limits the scalability of the platform.</td>
</tr>
<tr>
<td>1.2</td>
<td>Mesh partitioned for parallel computation using Gmsh [1]. The mesh has been split into four partitions algorithmically, with the intent of minimizing the number of boundary edges between each partition. This minimization lowers the communications overhead incurred when executing a CFD simulation on the mesh.</td>
</tr>
<tr>
<td>1.3</td>
<td>Mesh from a CFD solution of the common driven cavity problem on an unstructured grid. The mesh was generated using Gmsh [1], and is composed of triangles and quadrilaterals on the interior, with one dimensional cells all around the boundary.</td>
</tr>
<tr>
<td>1.4</td>
<td>Velocity field from a CFD solution of the common driven cavity problem on an unstructured grid. The solution was obtained using a code written by the author and is used only as an example of CFD results. The color corresponds to the velocity magnitude at each cell center and the lines portray both the direction and magnitude of the velocity.</td>
</tr>
<tr>
<td>1.5</td>
<td>Pressure field from a CFD solution of the common driven cavity problem on an unstructured grid. The solution was obtained using a code written by the author and is used only as an example of CFD results. The color corresponds to the relative pressure at each cell center.</td>
</tr>
<tr>
<td>1.6</td>
<td>General transported quantity field from a CFD solution of the common driven cavity problem on an unstructured grid. The solution was obtained using a code written by the author and is used only as an example of CFD results. The color corresponds to the magnitude of the quantity ( \phi ) at each cell center.</td>
</tr>
</tbody>
</table>
2.1 Storage topology of a modern computer cluster in two configurations. Configuration (a) shows the traditional CFD approach to I/O, where each node writes a separate file to disk. These files must be combined to obtain the full solution in a process which can be very time consuming. Configuration (b) shows the object of this work, being a CFD I/O method in which the partial solution from each node is written into the appropriate location within a single solution file for the whole problem.

2.2 Diagram depicting the hierarchical nature of a CGNS file. Data is broken into groups and sub-groups, with the location and type of a group in the hierarchy controlling how the data is interpreted.

4.1 Software stack used by the pCGNS library. Layers are ordered from most abstract at the top to least abstract at the bottom. Each layer relies on lower layers to implement functionality. This allows higher layers to be independent of the implementation of lower layers. Thus, as long as the MPI implementation used can take advantage of the file system on a machine, HDF5 and pCGNS will also be able to use that file system.

5.1 Topology of the computing clusters and network attached storage at Utah State University. A single high speed switch connects all the storage to the clusters, and serves as the I/O backbone for Wasatch. The network connections between the Wasatch nodes and the main switch are composed of pairs of 1 gigabit Ethernet connections bonded to form a 2 gigabit connection for each node. The connection between the Panasas storage (/panfs) and the switch is composed of four 10 gigabit Ethernet connections.

5.2 Data distribution scenarios for benchmarking. Part (a) shows two zones on one node. Part (b) shows one zone per node on four nodes. Part (c) shows four zones on two nodes. Part (d) shows two zones on four nodes. Part (e) shows four zones on four nodes with zones spread across multiple nodes.

5.3 Performance of the MPI-IO library on Wasatch using the IOR benchmark. IRead stands for independent read, CWrite for collective write, and so on. The hints recommended by Panasas are active to ensure best performance.

5.4 Performance of the HDF5 library on Wasatch using the IOR benchmark. IRead stands for independent read, CWrite for collective write, and so on. The hints recommended by Panasas are active to ensure best performance.

5.5 Performance of HDF5 on Wasatch using collective writes only, on a 256 MB data set. The hints recommended by Panasas are inactive to give results which are comparable to the pCGNS benchmarks which were not run with any hints activated.
5.6 Results of pCGNS benchmark (a). The larger file sizes experience approximately the same bandwidth, while the smaller sizes enjoy the benefits of buffered file I/O. The speedup of all sizes of file is approximately one, which demonstrates good efficiency in the pCGNS library, and all layers beneath it.

5.7 Results of pCGNS benchmark (b). Large file sizes perform better than smaller ones. The largest file size performs four times better in parallel I/O than the serial I/O, but performance nearly levels off after four nodes, possibly due to the topology of the storage at USU.

5.8 Results of pCGNS benchmark (c). Two zones are written by each node while keeping the file size constant, causing less and less data to be written with each additional node. The performance is still acceptable, but does level off after four nodes.

5.9 Results of pCGNS benchmark (c). Four zones are written by each node while keeping the file size constant, causing less and less data to be written with each additional node. The performance is no longer an improvement over the serial case with the smallest data size, although larger files still perform acceptably.

5.10 Results of pCGNS benchmark (c). Eight zones are written by each node while keeping the file size constant, causing less and less data to be written with each additional node. The performance is no longer an improvement over the serial case with all but the largest data size, which barely performs better than the serial version.

5.11 Results of pCGNS benchmark (d). Multiple nodes share each zone, requiring HDF5 to sort out any possible data overlaps. This causes a severe slow down of the I/O operations. None of the tested file sizes outperformed the serial case.

5.12 Results of pCGNS benchmark (e). The most general of the benchmarks, type-e demonstrates the performance when multiple zones are written by multiple processes. This situation results from parallel partitioning independent of the zone arrangement of the domain.
Chapter 1

Introduction

1.1 Scientific Computing

Throughout the history of scientific progress and engineering development there has been a need to compute the values of mathematical expressions to obtain results from theoretical work. In the earliest days, these computations were carried out by hand. This later developed into specialized calculating aids such as the Arabic abacus, Japanese soroban, or the Russian s’choty [2]. While faster than the unaided mind, these methods were still just an aid to the user’s efforts. Modern computing machines started to appear in the early twentieth century and have continued to develop to this day [3].

1.1.1 Data Storage History

Since the dawn of the computer age, there has been the need to get information into computers and to then receive the results of the calculations. With the earliest mechanical computers, this was often done in the form of nobs or dials, which allowed the operator to set the inputs, perhaps select one of a small set of operations to be done, and then read the results. While this method worked, and was faster than doing the calculations by hand, it was still very cumbersome and dramatically slowed down the entire process. Later the development of punch cards significantly improved the situation, allowing vast amounts of data to be entered and processed; the results of the computation themselves often being punched onto cards. The fact that large amounts of data could be processed in addition to the ability to write results in a machine-readable format contributed to the punch card being a dominant method of I/O even into the 1980’s [3].

Later developments include the use of punched tape and magnetic tape storage, along with various lesser-used mediums. The real advancement toward modern computer storage came with the first hard drives, which only became affordable for common use in the 1970’s and 1980’s.
The hard disk offers the storage of vast amounts of data with the ability to access it at relatively high speeds [4]. The capacity of hard drives has continued to increase exponentially since their invention in the 1950’s, and it is this large capacity combined with their flexibility that has made them the primary form of data storage on modern computers, from the largest cluster to the smallest laptop [5].

1.1.2 System Level I/O

On modern computer systems, programs nearly always rely on an operating system of some sort to provide support facilities to the program, often done in the form of some abstraction of the hardware into a more universal model. This allows programs to be written for one piece of computing hardware, but run on any hardware that provides the same abstract interface. The abstract interfaces used generally come in the form of a C application programming interface (API). Many operating systems provide the POSIX [6] API by which a program can interface with the system in a predefined way. Abstract interfaces are also provided by some programming languages, such as Fortran, with the actual implementation of the abstraction layer being found within the compiler.

Using this model, when a program needs to make some sort of I/O, it makes a request of the operating system to accomplish this. Most operating systems use libraries to provide the APIs, which themselves access the operating system’s kernel directly. The kernel is the core of the operating system which manages the lowest level of interaction between the hardware and software. This is done through the use of drivers, which present an interface to the computer’s hardware in a form that the kernel can use. Many kernels include vast databases of drivers for commonly encountered hardware, but if the kernel does not have native support for a device, a new driver can generally be loaded into the kernel to allow for operation of the device [7].

The exact nature of the I/O need not be storage access. In the broadest terms, I/O is any interaction between the computer and the outside world. This includes keyboard input, mouse movements, screen display, printer access, network connections and many more forms of I/O. For CFD and other simulation codes, the most important I/O forms are storage and network access [5].
1.1.3 Hierarchical Data Format

Nearly all scientific computing produces data which needs to be analyzed after completion of the simulation. Often these data sets can be very large and quite complicated in structure, making storage and post processing of the data more difficult. To solve this problem there are many file formats which are specific to each discipline within scientific computing, but this solution creates a new problem of needing specialized tools to be written for every field of study. To resolve this problem, general purpose data storage file formats have evolved which can describe any sort of data. One such file format is the hierarchical data format, or HDF [8].

HDF implements a sort of file system within a file. Data is arranged in groups, which can be layered and nested just like directories on a file system [9]. The actual data can be stored in a variety of built-in data types, or new data types can be created for the needs of a specific program. Arrays of built-in and user defined types can also be saved. This simple but flexible arrangement allows HDF to describe many different types of data, ranging from historic stock data to CFD simulation results.

1.1.4 Supercomputing

The fundamental motivation for computing hardware has always been the improvement of computation time required to obtain some result. Many advances have been made in the field of computing, especially in the last century with the development of the modern transistor-based digital computer [3], but even with the significant computational power available to modern desktop computers the needs of the scientific and engineering communities continue to demand more computational power to solve the problems with which they work.

1.1.5 Vector Computers

Early efforts to build computers which met these unusually large requests for processing power generally revolved around building a single monolithic computer with more processing power. This led researchers in the field of computing, such as Seymour Cray, to design vector computers. A vector computer is based on the execution model of single instruction multiple data, or SIMD. These computers were designed with large vector computation registers capable of holding many floating
point values and running the same operation on all of them simultaneously [10]. The ramifications of this hardware design are fully grasped when one considers the frequency with which one encounters vector or array operations when developing numerical analysis software for simulation of physical phenomena [11].

1.1.6 Cluster Computing

By the late 1990’s the speed of commodity computers was closing the gap with special-purpose supercomputers on a per processor level. The main difference between the two types of computers largely became a matter of scale rather than processor design. This development paved the way for modern computer clusters to make their debut. Initially named after the first computer of this type, cluster supercomputers are often referred to as Beowulf clusters [10]. A cluster supercomputer is a set of “off the shelf” computers which work on a single problem as a group rather than individually [12]. Because the components for clusters are either identical, or at least similar to standard desktop computer components, the cost of clusters is lower than that of the traditional supercomputers. The expertise required to design a cluster is also significantly less, since most of the work entailed is networking the computers together, not designing new computer hardware [13].

Computers in a cluster are connected using a network of some sort, which is generally called the interconnect of the cluster. This network is used by the various computers in the cluster, called nodes, to communicate information to each other about the solution process being executed. The kind of data that must be sent back and forth between the nodes of a cluster is highly dependent on the problem being solved. In CFD simulations, the data is generally the current flow solution along the domain boundary between sections of the mesh owned by different processes on the cluster [14, 15].

While any network can be used as the cluster interconnect, specialized network hardware and protocols have been developed to meet the needs of cluster computers. Some examples of these are Myrinet [16–19] and Infiniband [20–23]. Specialized networks are commonly used in clusters because the interconnect of the cluster is being used as an equivalent of the main data bus on a monolithic computer, and thus requires high speed performance for economical operation of the
machine [12, 24]. The data bus is a system on the mother board of a computer which carries information from one component of the computer to another. Any time data is accessed from memory, sent to the network, or accessed on storage, it needs to pass from one component to another via the data bus, and generally makes a stop at the CPU. All computers have a data bus, but not all buses are equal, nor are all routes through the bus of equivalent speeds.

Figure (1.1) portrays a cluster computer and a shared-memory computer. It can be seen that the cluster computer relies on its interconnect to send data from some processors to other processors, but that on each node of the cluster a data bus moves data from point to point. This is in contrast to the shared-memory computer which uses a single very high speed bus to move the data around the machine. The need to have a high speed, high bandwidth data bus greatly complicates the design of shared-memory computers and limits the scales on which they can be built. The path from the processors and memory to the storage of each machine is also quite different. The shared-memory computer’s data bus provides direct access from the CPUs and memory to the storage, whereas clusters often access a centralized storage system through a network separate from the primary interconnect.

1.2 Computational Fluid Dynamics

The field of computational fluid dynamics concerns itself with the simulation of fluid flow using computers. This process is non-trivial due to the complexity of the equations which must be solved and the numerical properties of the algorithms used to approximate these equations. These difficulties are compounded by the scale of most CFD problems.

1.2.1 Problem Formulation

Computational fluid dynamics codes generally work by solving some form of the differential equations which govern fluid flow. Equations (1.1) through (1.4) show those equations most often used as the governing equations by CFD codes, although there are many additional equations which may also be solved besides these. Some assumptions can be made to reduce the number of equations which must be solved, but such assumptions restrict the code to certain flow regimes, such as incompressible, inviscid, etc. [25].
Fig. 1.1: A generalized diagram of a cluster computer (Upper) and a shared-memory computer (Lower). The abbreviations used in the diagram are as follows: CPU=Central Processing Unit; NIC=Network Interface Card; HD=Hard Disk. The dashed line connecting all the components of each computer represents the primary data bus of the machine. The simplicity of the shared-memory computer architecture compared to that of the cluster is apparent, making shared-memory computers desirable when possible. Unfortunately, the performance required from the main data bus in a shared-memory computer limits the scalability of the platform.
The continuity equation, (1.1), enforces the conservation of mass during the simulation. In the case of steady-state incompressible solvers, it often also provides the coupling between velocity and pressure, since the incompressible assumption removes the physical relationship between them. The momentum equation, (1.2), is actually a vector equation, which provides one equation for the conservation of momentum in each dimension of the simulation. These equations enforce Newton’s second law of motion, and here are presented with the Newtonian fluid shear tensor assumption already made.

Normally only found in compressible flows or when heat transfer is of interest, the energy equation, (1.3), and the accompanying equations of state, (1.4), are used to couple the pressure and temperature to the density, as well as other material properties such as viscosity and thermal conductivity.

\[
\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_i} (\rho u_i) = 0 \quad (1.1)
\]
\[
\frac{\partial}{\partial t} (\rho u_i) + \frac{\partial}{\partial x_j} (\rho u_i u_j) = - \frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_i} \left[ \mu \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) + \delta_{ij} \mu' \frac{\partial u_k}{\partial x_k} \right] \quad (1.2)
\]
\[
\rho \left( \frac{\partial e}{\partial t} + u_i \frac{\partial e}{\partial x_i} \right) = \frac{\partial Q}{\partial t} + k \frac{\partial}{\partial x_i} \frac{\partial}{\partial x_i} T + \Phi \quad (1.3)
\]
\[
p = p(e, \rho) \quad \text{and} \quad T = T(e, \rho) \quad (1.4)
\]

### 1.2.2 Solution Algorithms

Many different approaches have been used to solve the equations governing fluid flow, but most of them can be identified as one of three commonly encountered forms. Finite difference methods (FDM) can be used to solve the equations with very high computational efficiencies, and bring the added benefit of being easy to understand and program [26]. Unfortunately, these methods are restricted to structured computational grids [27], although these grids may be deformed in the physical coordinates. Most early investigations in CFD used finite difference methods [28].

Finite volume methods (FVM) allow for fully unstructured meshes to be used in the solution process, but they are more complex to understand and program. They do have the desirable quality that each term in the discretized form of the equations can be directly related back to some physical...
meaning. This intuitive correlation is attractive to researchers in the field as it makes understanding the
codes easier than it would otherwise be. Most current commercial CFD codes use the finite
volume method [29].

Finite element methods (FEM) have recently become more popular in the field of computa-
tional fluid dynamics, but suffer from very abstract derivations being required to build a full CFD
solver. Like FVM, these methods can be used on arbitrarily complex meshes [27]. Unlike FDM and
FVM, finite element methods generally form the discretized equations in matrix form and solve the
system directly, whereas FDM and FVM usually use an in-place iterative solution which does not
require the full linear system to be expressed in matrix form [26].

1.2.3 Parallel Implementations

Because of the computation cost of solving most problems in CFD, it is normal for solvers
to be designed to take advantage of computing clusters or multiprocessor systems in order to ob-
tain solutions more rapidly. This is generally accomplished by partitioning the physical domain
of the problem into regions of approximately equal computational cost and then distributing these
partitions among the computing resources available as shown in Figure (1.2). During solution, the
processes which contain neighboring partitions of the domain pass solution information back and
forth between each other, so that the solution of the entire domain is computed [15]. This process
places large loads on the interconnect of computer clusters [30] and is one of the motivations for the
development of faster interconnects such as Infiniband or Myrinet.

Beyond the desire for additional performance, CFD codes are also parallelized to allow larger
problems to be solved than can be done with a single desktop computer. Since the domain of the
problem is distributed among the nodes of a cluster, no single node needs to store the entire solution.
This allows problems to be solved which are too large to fit on lesser machines.

1.2.4 Storage Concerns

Traditional CFD works by splitting the problem domain into smaller regions composing the
mesh or grid of the problem and applying some sort of discretization scheme to the governing
differential equations so as to satisfy those equations on each element or volume of the mesh. This
Fig. 1.2: Mesh partitioned for parallel computation using Gmsh [1]. The mesh has been split into four partitions algorithmically, with the intent of minimizing the number of boundary edges between each partition. This minimization lowers the communications overhead incurred when executing a CFD simulation on the mesh.
means that the field variables are stored as a finite set of sample points and interpolated between these known values. As a result, the mesh must be sufficiently refined to ensure accuracy, which causes the number of points at which the field variables must be stored to be very large.

Besides the size of the data sets used in CFD, the complexity of the data is also problematic for storage. Figures (1.3) through (1.6) show a sample CFD solution to the well-known driven cavity problem as computed by a code written by the author. The mesh is unstructured, which results in data structures needing to be stored which describe the location of each point, as well as the structure of each cell in the mesh. In this example, four field quantities are stored; those being the velocity component in the $x$-direction, the velocity component in the $y$-direction, the pressure, and an arbitrary parameter $\phi$, which is transported by the fluid. Each of these variables must be stored and somehow correlated to the mesh in order for the data to be meaningful.

The complexity of storing CFD is only made worse by the use of computer clusters in the solution process, since none of the nodes of the cluster have a copy of the entire solution in memory. This requires that the nodes somehow coordinate the reading and writing of solution data in such a way that it can be reassembled. Most often, this is accomplished by each process running on the cluster writing its own data in separate files. The files for each process are then recombined after completion of the simulation in order to obtain the final solution. Advancements in parallel I/O have removed the need for CFD codes to write one file per process, since multiple processes can now access the same file simultaneously [8].
Fig. 1.3: Mesh from a CFD solution of the common driven cavity problem on an unstructured grid. The mesh was generated using Gmsh [1], and is composed of triangles and quadrilaterals on the interior, with one dimensional cells all around the boundary.

Fig. 1.4: Velocity field from a CFD solution of the common driven cavity problem on an unstructured grid. The solution was obtained using a code written by the author and is used only as an example of CFD results. The color corresponds to the velocity magnitude at each cell center and the lines portray both the direction and magnitude of the velocity.
Fig. 1.5: Pressure field from a CFD solution of the common driven cavity problem on an unstructured grid. The solution was obtained using a code written by the author and is used only as an example of CFD results. The color corresponds to the relative pressure at each cell center.

Fig. 1.6: General transported quantity field from a CFD solution of the common driven cavity problem on an unstructured grid. The solution was obtained using a code written by the author and is used only as an example of CFD results. The color corresponds to the magnitude of the quantity $\phi$ at each cell center.
Chapter 2

Problem Description

2.1 Problem Introduction

A large portion of the work in fluid mechanics is now done in the form of computational fluid dynamics (CFD), in which a discretized form of the partial differential equations governing fluid flow are solved on a particular domain using a computer. CFD is used in design and optimization for many different applications. Such applications are so diverse as to include both jet fighters and bleach bottles. It would be difficult to find someone who had never interacted with something designed using CFD. In order to attain the accuracy needed for such projects, CFD requires significant computing resources, and as such is often completed with the aid of some sort of supercomputer. In the early days of CFD most supercomputers came in the form of monolithic vector computers which packed numerous processors into one single machine [10]. Modern supercomputers more often take the form of computing clusters, in which numerous desktop-like computers work together on solving the discretized equations [3]. Utah State University possesses several such computing clusters through the Center for High Performance Computing to facilitate CFD based fluid dynamics research, as well as other studies.

CFD demands not only a great deal of processing power, but also large amounts of data storage. Even a simple CFD simulation can produce gigabytes of information at a tremendous rate, and all this data must be stored somewhere so that the researcher can analyze the results of the simulation. This situation is worsened in the case of unsteady 3d simulations. The process of reading or writing data by a program is generally termed I/O, which stands for Input/Output. On the old vector computers this was mostly a matter of providing sufficiently fast hard-drives in the computer which could keep up with the I/O demands of the CFD code. On modern computer clusters however, the problem becomes more complicated. Because of the distributed nature of computer clusters, the domain of a CFD problem must be broken up into small pieces and scattered across the nodes of the
cluster. This means that each node in the cluster only has a small part of the whole problem [31]. When solutions need to be written to disk for later examination, there are several options. Firstly, each node can simply write its part of the problem to some local or network storage, and the various pieces can be reassembled after completion of the simulation. While this solution has the benefit of simplicity, it does not provide desirable performance, since the reconstruction process is lengthy, and in extreme cases can take longer than the duration of the simulation itself. The second option is for the nodes of the cluster to all write to one file on the storage system simultaneously. For this to be done the nodes must cooperate on the I/O in parallel. This method is harder to program, but should be able to provide nearly the same write-time performance as the first method, without the need to reconstruct the data after completion of the simulation. Implementation of this second form of clustered CFD I/O is the purpose of this work. An overview of the two I/O methods is provided in Figure (2.1).

Since the data written by CFD codes tends to be quite complex, specialized file formats have been developed to standardize the storage and processing of results obtained from simulations. One such file format is known as the CFD General Notation System (CGNS) [32]. This file format provides a standardized method of storing the geometry, configuration and solution of a CFD simulation, including support for time varying flow solutions. Such standardization is very important for CFD since it allows researchers and designers to share both the results of a simulation and the setup of the simulation in a single uniform format. Many organizations use or support the CGNS standard including Boeing, Onera, Fluent and others [33].

While any program which complies with certain specifications can write acceptable CGNS files, there is a standard library which can be used to write CGNS files from either the Fortran or C programming languages. This library, known as the CGNS Mid-level Library (MLL), currently only supports the older method of CFD I/O. It supports two different styles of CGNS file, the first being an older database-like file format called ADF, and the second using a scientific data storage library called HDF5. The plan for the CGNS file format and library is to migrate completely from ADF to HDF5 [34]. The HDF5 library itself already supports the low-level parallel I/O features needed by the proposed parallel version of CGNS [9]. It does this through the use of the MPI-IO library.
Fig. 2.1: Storage topology of a modern computer cluster in two configurations. Configuration (a) shows the traditional CFD approach to I/O, where each node writes a separate file to disk. These files must be combined to obtain the full solution in a process which can be very time consuming. Configuration (b) shows the object of this work, being a CFD I/O method in which the partial solution from each node is written into the appropriate location within a single solution file for the whole problem.
The most recent revision of the MPI standard has added specifications for parallel I/O, and has been implemented by the ROMIO developers at Argonne National Laboratory [35].

The purpose of this work is to augment the CGNS library so as to allow for parallel writing of HDF5 based files without the need for radical changes to the CFD codes themselves. The augmentation is accomplished in the form of a new library which initially implements a sub-set consisting of the most used features from the full CGNS standard. The new library is called pCGNS, for parallel CGNS. Unlike the current implementation of HDF5 support in the CGNS library, this new library uses the HDF5 API directly instead of accessing it through an emulation of the ADF API. This allows pCGNS to access the parallel I/O routines and abilities of the newer HDF5 library versions. It also simplifies the library’s internal structure.

2.2 Problem Background

The original CGNS library was conceived during the 1990’s to resolve some of the more important difficulties in transferring NASA technologies to industrial development. Because NASA and industry used many different file formats which were incompatible with each other, data and software exchange between NASA and industry was limited. The CGNS file format and library were introduced to solve these problems, and have been widely adopted by both government agencies and industrial entities around the world [33]. This adoption is largely due to the ability of CGNS files to express nearly any CFD simulation setup, and the work-flow simplification that ensues from its use [36].

Also in the 1990’s, a transition was occurring within the field of supercomputing. Large monolithic supercomputers such as those offered by Cray Computing were being replaced by lower cost computer clusters, often called Beowulf clusters when built out of “off the shelf” components and running a Linux distribution [10]. While this transition was driven by lower costs and higher possible performance of the new cluster computers, they also presented new problems. A computer cluster requires a high speed network which allows the nodes of the cluster to communicate one with another. Such a network is often termed the cluster interconnect. Any network connection can be used in this capacity, but specialized solutions have been developed such as Myrinet [16–19] and
Infiniband [20–23].

Another difficulty which has arisen from the transition to clustered supercomputing is the implementation of high speed data storage. Since cluster computers often separate the computation nodes from the storage nodes, generally in the form of a commercial or custom network attached storage (NAS) system [37], the rate at which data can be transferred from memory on a computation node to a storage node is therefore limited by the speed of the network connection between these nodes and the NAS. Additionally, the storage nodes and computation nodes are sometimes not connected by the high speed interconnect of the cluster, since many clusters may share the same storage. This is further limited by the fact that the file servers used in the early cluster computers only supported a single client writing at a time for each file. Thus multiple processes across the cluster could not write to the same file simultaneously. This caused many simulation codes to adopt the practice of each process writing to a separate file during parallel operation, with the intent that the files would be reconstructed after completion of the simulation [31].

As a first step to solving this problem, parallel file systems have been developed for use on computer clusters and network attached storage such as Lustre [38] and PVFS [39]. These file systems allow multiple clients to open and modify a single file simultaneously. This development allows for a parallel simulation running on a computer cluster to write all of its data to a single file without the need for reconstruction. The need for such an operation is created by the way in which CFD codes distribute the computational work on a cluster, which is to partition the computational domain of the problem and distribute the various physical regions in the domain among the processes running on the cluster. A partitioned example mesh is shown in Figure (1.2).

The problem which this effort works to address is the lack of a method for writing CGNS files in parallel. Since CGNS is a format specifically aimed at CFD, and many CFD codes run in parallel on computing clusters, the lack of support for parallel file systems in the main CGNS library is an obstacle to its goal of being a universal data exchange format for the field of CFD. By implementing the parallel extension to CGNS, this impediment to performance and flexibility for CFD codes will be eliminated by allowing them to take advantage of the I/O performance now available to cluster computers through the use of parallel file systems on the network attached storage.
2.3 Computational Fluid Dynamics General Notation System

As stated previously, CGNS is a standardized method of notating the setup and results of a CFD case originally designed to facilitate collaboration between NASA and industry [40]. The CGNS standard can be separated into two components. The first component is the set of concepts that allow the case to be described in a fashion generic enough to suit many purposes, but still specific enough to include all relevant data. This is done using the standard interface data structures (SIDS). The second component of the CGNS standard governs how the SIDS are expressed in a file. This also includes a reference library capable of writing compliant files, called the Mid-level Library (MLL).

2.3.1 Standard Interface Data Structures

The SIDS describe a CFD case by using a hierarchy of nodes, with each node defining some quality incident to all the nodes subordinate to it. By doing this, all the data is organized by scope automatically. Most important to the current work is the organization of the mesh description and the corresponding solution data.

Figure (2.2) shows the hierarchy of an example CGNS file. The problem domain is broken up into bases, and then zones, although most files only contain a single base and many only contain a single zone as well. The choice of how to split up the domain into bases and zones is not defined by the CGNS standard but rather left up to the user [41].

It can be seen that each zone contains its own set of coordinates and solutions. The zone node contains the number of points and cells which define its portion of the domain, so the relationship between the solution data and coordinate data is known implicitly.

2.3.2 Advanced Data Format

The first file format used to express the CGNS standard was the advanced data format (ADF) which sees heavy use by NASA and the aerospace industry. ADF stores data in a hierarchical manner, making it a natural choice for the implementation of the CGNS standard. A mapping between the CGNS SIDS and ADF data structures is provided which allows any who wish to write a program which would write CGNS compliant files in the ADF container format. However, because of the complexity of this task most software supporting CGNS does so through the MLL.
Fig. 2.2: Diagram depicting the hierarchical nature of a CGNS file. Data is broken into groups and sub-groups, with the location and type of a group in the hierarchy controlling how the data is interpreted.
2.3.3 Hierarchical Data Format version 5

HDF5 also describes data in a hierarchical manner, and mappings have been defined for the use of HDF5 as the container format for CGNS [42]. Not only is it desirable to have multiple back ends to ensure the generality of the standard, but the newest versions of HDF5 support parallel I/O through MPI-IO, including collective I/O, which holds the potential for performance improvements and the ability to write larger data-sets directly [43].

2.3.4 Mid-level Library

The Mid-level Library provides C and Fortran interfaces with which applications can read and write standards-compliant CGNS files without the need to use ADF or HDF5 directly. Unfortunately, since CGNS initially only supported the ADF container format, the HDF5 support in the MLL is provided by an HDF5 to ADF compatibility layer which has been coded into the MLL. This compatibility layer ensured feature parity between the two back ends, but now prevents the parallel I/O abilities present in HDF5 from being accessed.
Chapter 3

Literature Review

3.1 Previous Work

Considerable research has been done on data access for scientific applications. The work has focused on data I/O performance and data management convenience. Three projects, MPI-IO, HDF5 and parallel netCDF (PnetCDF) are closely related to this research.

MPI-IO is a parallel I/O interface specified in the MPI-2 standard. It is implemented and used on a wide range of platforms. The most popular implementation, ROMIO [44] is implemented portably on top of an abstract I/O device layer [35,45] that enables portability to new underlying I/O systems. One of the most important features in ROMIO is collective I/O operations, which adopt a two-phase I/O strategy [46–49] and improve the parallel I/O performance by significantly reducing the number of I/O requests that would otherwise result in many small, noncontiguous I/O requests. However, MPI-IO reads and writes data in a raw format without providing any functionality to effectively manage the associated metadata, nor does it guarantee data portability, thereby making it inconvenient for scientists to organize, transfer, and share their application data.

HDF is a file format and software developed at NCSA for storing, retrieving, analyzing, visualizing, and converting scientific data. The most popular versions of HDF are HDF4 [50] and HDF5 [8]. Both versions store multidimensional arrays together with ancillary data in portable, self-describing file formats. HDF4 was designed with serial data access in mind, whereas HDF5 is a major revision in which its API is completely redesigned and now includes parallel I/O access. The support for parallel data access in HDF5 is built on top of MPI-IO, which ensures its portability. This move undoubtedly inconvenienced users of HDF4, but it was a necessary step in providing parallel access semantics. HDF5 also adds several new features, such as a hierarchical file structure, that provide application programmers with a host of options for organizing how data is stored in HDF5 files. Unfortunately, this high degree of flexibility can sometimes come at the cost of high
performance, as seen in previous studies [51, 52].

Parallel-NetCDF [53] is a library providing high-performance I/O while still maintaining file-format compatibility with Unidata’s NetCDF [54]. In the parallel implementation the serial netCDF interface was extended to facilitate parallel access. By building on top of MPI-IO, a number of interface advantages and performance optimizations were obtained. Preliminary test results show that the somewhat simpler netCDF file format coupled with a parallel API combine to provide a very high-performance solution to the problem of portable, structured data storage.

With these developments in the field of parallel I/O it is unsuprising that this is not the first time that a parallel extension to CGNS has been proposed. Hauser and Pakalapati previously worked on a possible parallel extension to the CGNS Mid-level library, but at the time the HDF5 library did not support collective I/O [43, 55]. This forced their efforts to use independent I/O, which is anticipated to be slower than collective [9, 56]. Their work also did not provide the same level of compatibility with the main CGNS library as the current effort.

3.2 Parallel Input/Output

With the transition to cluster-based supercomputing, the parallelization of all processes involved in the solution of scientific computational problems has become a major concern in the industry. This includes the need for high-speed parallel access to the file systems mounted by the nodes of a cluster, as well as a unified means of coordinating the work of various nodes in a cluster to solve a single problem.

3.2.1 Message Passing Interface Standard

The switch from monolithic vector supercomputers to cluster computers created the need to coordinate the solution of a problem on more than one machine at a time. The problem simplifies into the need for a standardized means of transferring data back and forth from one node to another in the cluster. Traditional networking APIs can be used in this manner, and still drive the communications of certain parallel programs such as DaVis from LaVision [57]. The needs of scientific parallel codes are very specific, however, and do not fit well with the more generic networking APIs available. This has led to the development of specialized means of communication between nodes
of a cluster [58]. One earlier development was the parallel virtual machine (PVM) library and runtime [59]. Among other things, it provides a message passing interface which allows one process running on the machine to easily send information to another in a standardized way.

The concept of message passing proved to be an effective paradigm in which to program cluster computers. This success led to the development of a standard message passing interface (MPI) defined by the MPI Forum [60] and implemented by various vendors and open source groups [61,62]. Since all implementations of MPI adhere to the standard, programs can be written against the MPI standard and then compiled and run on any computer which has an MPI implementation. Because of this open nature and the effectiveness of the message passing concept for parallel programming, MPI has risen to be the primary means by which programs are written to run on a computer cluster, or even on symmetric multiprocessor machines such as modern multi-core computers.

The MPI standard defines a library which provides the MPI API. This API supplies the programmer with numerous communications subroutines which allow the processes running on a supercomputer to pass information back and forth. Some of the routines are quite simple such as \texttt{MPI\_Send}, which sends the data contained in a memory buffer from one process to another. The receiving process must call a matching \texttt{MPI\_Recv}, which receives the sent data into a memory buffer. More complex communications are also supported, such as operations which find the average, sum, or max of a set of memory buffers on various processes across the machine. Using these routines, the program can ensure that data passes through the entire machine as if it were one computer, despite the fact that it may be composed of many discrete computers scattered across large geographical distances.

MPI provides a run-time component as well as the main library. This run-time handles the spawning of all the processes for an MPI job and provides the library with critical execution information such as how many processes are running in the job and the rank of a process relative to the group. The run-time component also implements the fundamental code which actually executes the message passing. Since many cluster computers are composed of nodes which themselves are symmetric multi-processor machines [63], the best way to send data from one process to another greatly depends on the physical path from the processor executing the send routine and the processor exe-
cuting the receive routine. Most implementations of MPI are intelligent about how a message is sent from one process to another. In the case of two processes which are executed on the same physical node, the message is generally passed through a shared memory interface [61]. This can be faster than the interconnect of the cluster, depending highly on the code being executed, and allows MPI to be used as the parallel programing interface for multi-processor machines as well as clusters.

3.2.2 Network Attached Storage

A network attached storage (NAS) unit is a computer system attached to a network for the purpose of providing storage to other clients of the network [37]. While a NAS can be built out of commodity hardware just like the nodes in a cluster, there are commercial solutions which can be purchased pre-built. Being a complete computer system, a NAS contains a CPU, memory, and other features common to servers. What make a NAS capable of fulfilling its purpose is the operation of file server daemons running on the machine. These allow clients on the network to access the files stored on the NAS’s internal hard drives.

3.2.3 Message Passing Interface-I/O

With the rise of parallel file systems on network attached storage or even integrated into the cluster itself, the need arose for a standardized way to access these file systems, so that simulations could be written which would run on any system with a simple recompile. To this end, the MPI standard was extended in version 2 to include file access routines which allow for parallel storage I/O. These routines are designed to be generic enough to fit any of the file systems that can be used as a back end, including the old standard serial routines if no parallel system is available. In general, implementations of MPI need to write interfaces for each parallel file system that they will support, and then the system administrator will enable the appropriate back ends when MPI is compiled for a new machine [35, 60].

3.2.4 Parallel File Systems

NAS systems are commonly used on large networks to provide a central location for storing shared data. As such, the concept of a using a NAS is well adapted to cluster computing since all
the nodes working on a problem need access to the same sets of files. Generally a NAS provides file servers such as Server Message Block (SMB) or Network File System (NFS) to allow clients access to the stored data. These protocols do not allow for parallel access to files, however. To resolve this problem, several parallel file systems have been developed which can either run on a NAS or set of NAS systems working together or can be run directly on the nodes of a cluster equipped with hard disks in each node and special daemons running on the nodes to provide access to the data.

3.2.4.1 Contiguous vs Dis-contiguous I/O

Most common file systems, including parallel file systems, represent a file as a one dimensional array of bytes. Assuming that the data stored on a hard drive is not fragmented, this model is actually very close to the true nature of the data’s organization on the drive. A data-set which occupies every byte in a contiguous region of the file is called contiguous data. When data is contiguous, it can be read and written very efficiently by modern hardware, which takes advantage of memory-buffered reads and writes to speed up these operations.

Dis-contiguous data-sets are scattered throughout a region of a file. Even if the organization of the data is orderly, dis-contiguous data cannot be written nearly as efficiently by storage systems.

In serial computing, the difference between contiguous and dis-contiguous data is not a serious problem since ensuring that a data-set is contiguous in storage is often as simple as ensuring that it is contiguous in memory. In parallel computing, when a domain has been split between multiple processes, data which is contiguous in memory is still dis-contiguous in storage.

3.2.4.2 Collective vs Independent I/O

The semantics of MPI-IO define two different types of I/O, those being collective and independent. Independent I/O is characterized by each process in an MPI communicator accessing the file independently from the other processes in the communicator. This reduces the amount of extra communication required to manage the I/O, since each process accesses the file without regard for the activities of the other processes.

Collective I/O requires that every process in the communicator participate in the I/O operation. While this requires greater communication between the processes to synchronize the operation, it
allows MPI to use the high speed interconnect and user defined data descriptions to aggregate the data as it is sent to the data storage system. This can dramatically increase the total bandwidth of the operation since the data is rearranged into a more contiguous form [43, 49].

### 3.2.4.3 PVFS

Currently being developed by at the Parallel Architecture Research Laboratory at Clemson University, the Mathematics and Computer Science Division at Argonne National Laboratory, and the Ohio Supercomputer Center, PVFS is a parallel file system designed to be as close to standard Unix file systems as possible while still providing the performance and parallel access capabilities as other parallel file systems [39, 64, 65].

### 3.2.4.4 Lustre

Originally developed at Carnegie Mellon University, the Lustre file system was designed with Linux-based scientific computing clusters in mind from the beginning. A feature of Lustre common to all object-based distributed file systems is the separation of the file meta data from the actual file data. The meta data of a file includes its name, path, creation time, and other properties, but is generally only needed when opening a file. This way the meta data servers can direct the clients to other servers which actually store the file’s data. Lustre is highly scalable and has seen use on the Blue Gene installation at Lawrence Livermore National Laboratory, one of the most powerful computers in the world [38, 66]. Additionally, it has been demonstrated that the Lustre file system can accommodate storage systems spread across very large geographical distances [38]. Lustre has been shown to be more performant than PVFS [67].

### 3.2.4.5 GPFS

Introduced in IBM’s own custom Unix variant AIX, the General Parallel File system is now supported on the AIX, Linux and Windows platforms. Like Lustre, GPFS splits the meta data from the file content, allowing for higher performance. The file system also provides complete POSIX compliance [68].
3.2.4.6 Panasas

The Panasas storage products are good examples of commercially available cluster storage solutions. Providing both the needed software and hardware components, these systems employ a proprietary file system tightly integrated with the storage nodes’ operating systems. The clients interface with the storage through kernel modules and libraries, which are supported by some implementations of MPI-IO [69].

3.2.5 Hierarchical Data Format version 5

With the advent of parallel I/O and other changes to the needs of a standardized data format, the HDF file format released a new revision called HDF5. The later versions of the HDF5 library, especially 1.8.x, provide support for parallel I/O by making use of MPI-IO as a back end. This allows HDF5 to support any parallel file system which the MPI implementation can handle. The HDF5 library also simplifies the use of parallel I/O and allows the programmer to focus on the data being described and let HDF5 determine the most efficient way to access the parallel storage [43].
Chapter 4
Parallel Implementation for the CFD General Notation System

The reimplementation of CGNS, directly using HDF5 as the container file format with the intent of parallel access to files is the purpose of the pCGNS library. Currently it is intended as a companion library to the Mid-level Library, since pCGNS only supports the most important and I/O-bound of the operations needed to write an entire CGNS file. The library is written in C and complies to the C99 standard. This presents many syntactic advantages over traditional C which greatly simplify the process of writing the library [70]. At present there is no Fortran interface to the library, but the addition of such an interface would not present much difficulty, especially in light of the automatically generated interface present in the main CGNS library.

4.1 Design

Figure (4.1) shows the software stack used to implement the functionality of the library. The Figure shows the layers of software libraries, with calls only made from one adjacent layer to another. Thus, pCGNS is entirely dependent on HDF5 and MPI, and completely ignorant of any particular file system. This allows the lower layers of the system to be swapped out from one system to the next without the need to redesign pCGNS.

Internally, the library is composed of two primary source files. The first, pcgns_util.c contains utility functions which are used repeatedly by the main library. These functions are not intended to be called by a program but rather are to be called from other parts of the library only. The utility functions implemented in pcgns_util.c are either used to maintain the internal state of the library, which uses several global variables to keep track of open files and details about those files, or the functions exist to streamline certain commonly executed operations using HDF5. These operations include the creation of SIDS nodes within the file, and the reading and writing of simple data types. The presence of global variables in the library precludes the possibility of calling the library from a
Fig. 4.1: Software stack used by the pCGNS library. Layers are ordered from most abstract at the top to least abstract at the bottom. Each layer relies on lower layers to implement functionality. This allows higher layers to be independent of the implementation of lower layers. Thus, as long as the MPI implementation used can take advantage of the file system on a machine, HDF5 and pCGNS will also be able to use that file system.

multi-threaded program such as one created using p-threads or OpenMP, as this would corrupt the internal state of the library. Resolution of this issue would require a significant break from the MLL API, which would discourage adoption of this library by existing users of CGNS.

The second primary source file, pcgnslib.c implements the public interface to the library, or in other terms, this file actually implements the library’s API. These functions are intended to be called by external programs to write CGNS files. Most of these functions rely on either HDF5 or pcgns_util.c for functionality. Calls to MPI are present in both portions of the library.

The implementation details of the library, including the full annotated source code and structure diagrams are available in Appendix (A) of this document.

4.2 Features

Table (4.1) shows the functions defined in pcgnslib.h and which constitute the pCGNS API as it currently exists. The functions chosen to be implemented in the initial version of the library were selected on the basis of two criteria. The first was the need for the function to provide the most basic functionality of the library. The second criteria was the possible benefit from parallelization of the function. The result of using these priorities is that the pCGNS library can write the basic components of a CGNS file needed for the standard CGNS tools to function, but still provides
## General File Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cgp_open</code></td>
<td>Open a new file in parallel</td>
</tr>
<tr>
<td><code>cgp_base_read</code></td>
<td>Read the details of a base in the file</td>
</tr>
<tr>
<td><code>cgp_base_write</code></td>
<td>Write a new base to the file</td>
</tr>
<tr>
<td><code>cgp_nbases</code></td>
<td>Return the number of bases in the file</td>
</tr>
<tr>
<td><code>cgp_zone_read</code></td>
<td>Read the details of a zone in the base</td>
</tr>
<tr>
<td><code>cgp_zone_type</code></td>
<td>Read the type of a zone in the base</td>
</tr>
<tr>
<td><code>cgp_zone_write</code></td>
<td>Write a zone to the base</td>
</tr>
<tr>
<td><code>cgp_nzones</code></td>
<td>Return the number of zones in the base</td>
</tr>
</tbody>
</table>

## Coordinate Data Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cgp_coord_write</code></td>
<td>Create the node and empty array to store coordinate data</td>
</tr>
<tr>
<td><code>cgp_coord_write_data</code></td>
<td>Write coordinate data to the zone in parallel</td>
</tr>
</tbody>
</table>

## Unstructured Grid Connectivity Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cgp_section_write</code></td>
<td>Create the nodes and empty array to store grid connectivity for an unstructured mesh</td>
</tr>
<tr>
<td><code>cgp_section_write_data</code></td>
<td>Write the grid connectivity to the zone in parallel for an unstructured mesh</td>
</tr>
</tbody>
</table>

## Solution Data Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cgp_sol_write</code></td>
<td>Create the node and empty array to store solution data</td>
</tr>
<tr>
<td><code>cgp_sol_write_data</code></td>
<td>Write solution data to the zone in parallel</td>
</tr>
</tbody>
</table>

## General Array Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cgp_array_write</code></td>
<td>Create the node and empty array to store general array data</td>
</tr>
<tr>
<td><code>cgp_array_write_data</code></td>
<td>Write general array data to the zone in parallel</td>
</tr>
</tbody>
</table>

## Queued I/O Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>queue_slice_write</code></td>
<td>Queue a write operation to be executed later</td>
</tr>
<tr>
<td><code>queue_flush</code></td>
<td>Execute queued write operations</td>
</tr>
</tbody>
</table>

Table 4.1: API of the pCGNS library as implemented in `pcgnslib.c` and declared in `pcgnslib.h`. Software using the library to access CGNS files in parallel must do so using the routines listed here.

routines for parallel file access on the largest of the data sets that need to be written to the file. Any extra data or properties that may need to be written to the file can still be added using the MLL after the file has been closed with pCGNS.
Chapter 5

Results

After implementation of the pCGNS library, several benchmarks of the library were run on Wasatch, the newest computer cluster at Utah State University. The benchmarks run were selected to explore the performance of the library under conditions which would be typical of the file access patterns generated by CFD codes. In addition to the benchmarks of the pCGNS library, another benchmark program has also been run, called IOR, which provides a measurement of the dis-contiguous parallel I/O performance of a cluster when using both the MPI-IO and HDF5 interfaces. This second set of benchmarks provides an upper limit against which the performance of pCGNS can be compared.

5.1 Hardware Setup

Figure (5.1) shows the setup of the HPC computer systems at USU. The older cluster, Uinta, on which much of the developmental work and testing was done, is shown with its 1 gigabit connection to the central Procurve switch. An odd feature of Uinta is the non-uniform access to the storage systems. This not only complicates the process of benchmarking, since which nodes the job runs on affects the results of the job. The newer cluster, Wasatch, has a connection from each node directly to the Procurve switch. This connection is actually two 1 gigabit Ethernet connections bonded together to form a single connection. Not only does this put every Wasatch node at an equal position with regard to storage, but it also provides a high-speed direct link to the switch.

The storage used in these benchmarks is a parallel storage solution provided by Panasas, mounted at /panfs. All of the files used by the program during the benchmarking process are stored on /panfs, which is composed of four shelves of storage servers. Each shelf is connected to the Procurve switch using a 10 gigabit Ethernet connection.
Fig. 5.1: Topology of the computing clusters and network attached storage at Utah State University. A single high speed switch connects all the storage to the clusters, and serves as the I/O backbone for Wasatch. The network connections between the Wasatch nodes and the main switch are composed of pairs of 1 gigabit Ethernet connections bonded to form a 2 gigabit connection for each node. The connection between the Panasas storage (/panfs) and the switch is composed of four 10 gigabit Ethernet connections.
5.2 Benchmark Selection

The selection of benchmarks to be run using both the IOR benchmarking tool and the pCGNS library with its related test programs was accomplished using several criteria and goals as outlined below.

5.2.1 IOR

Two sets of benchmarks were selected to be run using the IOR benchmarking tool. The first set is intended to characterize both independent and collective read and write I/O performance on Wasatch. These benchmarks were selected to give an idea of the complete I/O capabilities of the new machine.

A second and smaller benchmark was selected to be run in collective write mode with only the HDF5 API, as a direct comparison with pCGNS.

5.2.2 pCGNS

Five categories of benchmark were selected to be run on the pCGNS library. These categories are shown in Figure (5.2). Each scenario of data distribution models a general method of splitting up data among the processes of a parallel program. The scenarios will be hereafter referred to by the designation each scenario bears in the figure. In all of the benchmarks, the file size is kept constant, since during practical use of the library the file size will generally not be a function of the number of nodes used for a simulation, but rather a function of the simulation itself.

The first of these, the type-a benchmark, explores the performance of the library when multiple zones are stored on a single node. This is the only benchmark in which multiple processes are run on a single node. Normally this is avoided since having multiple processes on a node forces those processes to share a single network connection, thus affecting the measurement of the parallel performance of the library. In the type-a benchmark, however, this limitation is desired since it allows the single process with multiple zones to have access to the same storage bandwidth as the aggregate bandwidth of multiple processes on the node.

The type-b benchmarks measure the performance of pCGNS in a configuration very close to the benchmarks run by IOR. In this scenario, each node runs a single process, and each process only
writes a single zone. Because each zone has its own data array in the file, and each zone is written by a single process, each process writes to a single data array in the file.

The type-c benchmarks test the ability of the library to write multiple zones from each node in parallel. Because the file size is kept constant, more zones per node result in smaller data arrays written to the file in each I/O operation. Since this is expected to affect performance, multiple cases are run with different numbers of zones per node.

Type-d benchmarks test the ability of the library to write a single zone from multiple nodes. This causes multiple processes to access a single data array simultaneously, which is quite different from multiple processes accessing multiple data array. In this scenario, each node only writes to a single zone, but each zone is written to by multiple processes.

The last scenario, type-e, tests the most general access pattern, in which multiple processes access each zone and multiple zones are accessed by each process. This access pattern would result from the zone definitions and parallel partitioning processes being completely independent one from another.

5.3 Benchmark Results
Fig. 5.3: Performance of the MPI-IO library on Wasatch using the IOR benchmark. IRead stands for independent read, CWrite for collective write, and so on. The hints recommended by Panasas are active to ensure best performance.

5.3.1 IOR

Figures (5.3) through (5.4) detail the performance of USU’s newest cluster Wasatch with regard to accessing a single file from multiple processes using several different APIs via the IOR benchmark software.

Figure (5.3) shows the IOR benchmark results when using the MPI-IO API to read and write files in both collective and independent modes. It can be seen that there is not much difference between the collective and independent I/O when using the MPI-IO interface. The fact that the four-processor job enjoyed much faster write speeds is most certainly a result of buffered I/O on the part of the file system. This is accomplished by the kernel module used to interface with the storage keeping the data in memory while it waits finish writing, but allowing the benchmark program to continue, thus giving the perception of much faster I/O. It can also be seen that read speeds are significantly faster than write times, which is normal for these kinds of I/O operations.

The plot presented in Figure (5.4) displays the results of the same benchmark but uses the
Fig. 5.4: Performance of the HDF5 library on Wasatch using the IOR benchmark. IRead stands for independent read, CWrite for collective write, and so on. The hints recommended by Panasas are active to ensure best performance.

HDF5 interface. The most important difference between the MPI-IO and HDF5 results is the markedly improved performance provided by the HDF5 library, in both collective and independent I/O. The HDF5 library has received considerable work to provide high performance by optimizing I/O based on the structure of the HDF5 file to which the data is being written. Also of note is the larger difference between the collective and independent I/O modes. It would appear that the independent routines are better-optimized than the collective ones.

The last IOR benchmark was run to get a candidate for direct comparison with the performance of the pCGNS library. Since pCGNS currently only operates in collective write mode, that was the only mode tested in this benchmark. The data size was chosen to be 256 MB per process, since the pCGNS tests are run data sets of approximately this size. Figure (5.5) shows the results, which demonstrate a sharp drop in performance when the library switches into true parallel operation.
Fig. 5.5: Performance of HDF5 on Wasatch using collective writes only, on a 256 MB data set. The hints recommended by Panasas are inactive to give results which are comparable to the pCGNS benchmarks which were not run with any hints activated.
5.3.2 pCGNS

All benchmarks were run five times, and the line shown in each plot is the average of the runs. Error bars are also shown, based on the standard deviation of the bandwidths. The speedup plots are based solely on the averaged data, and are computed at the bandwidth for a particular number of processes or nodes divided by the bandwidth for a single process or node.

The results of the type-a benchmarks are summarized in Figure (5.6). All of the processes in this case are on the same node, as the purpose of this benchmark is to gage the efficiency of the library itself, with no real parallel I/O advantages gained by using more nodes. While the various sizes of file did affect the overall bandwidth of the writes, with larger files the bandwidth is relatively constant between 65 and 70 MB/s. The smaller files write faster because of the buffering inherent in the file system. The results are actually promising, since the speedup of all sizes of file are very close to one. This means that the bandwidth is being limited by the network connection between the storage and the node doing the writing, not by some inefficiency in the library.

Figure (5.7) shows the results of the type-b benchmark, which investigates the parallel performance of the library in optimal conditions. In this case, each node writes a single zone to the file, with the size of the file being held constant. The plot demonstrates that the performance of the library is generally better with larger data sets. This is to be expected, since the larger data sets cause the time spent on inter-process coordination to be smaller relative to the total time of program execution. Also unsurprisingly, the performance worsens with more nodes when transferring small amounts of data, since each node is writing less and less data, which is less efficient.

From the speedup plot, it is seen that near-optimal performance is maintained using up to four nodes on the largest case. This is perhaps a result of the four shelves which compose the Panasas NAS. Yet again, the lower performance with smaller file sizes is expected. The best performance is four times faster than the equivalent operation using serial I/O, thus validating the possibility for performance improvement using parallel I/O.

The type-c benchmarks include more data than the others, since they were run with multiple numbers of zones per node in an attempt to measure the performance resulting from aggressively breaking up the computational domain. The results of these runs are summarized in Figures (5.8)
Fig. 5.6: Results of pCGNS benchmark (a). The larger file sizes experience approximately the same bandwidth, while the smaller sizes enjoy the benefits of buffered file I/O. The speedup of all sizes of file is approximately one, which demonstrates good efficiency in the pCGNS library, and all layers beneath it.
Fig. 5.7: Results of pCGNS benchmark (b). Large file sizes perform better than smaller ones. The largest file size performs four times better in parallel I/O than the serial I/O, but performance nearly levels off after four nodes, possibly due to the topology of the storage at USU.
through (5.10). When compared to the performance of the library in the type-b benchmarks, the largest file size with the fewest zones compares favorably, although it is slower. The speedup peaks at just under three times the performance of the serial code, and still levels off around four nodes. As the number of zones increases, however, the performance rapidly deteriorates. This is largely due to the phenomenon of smaller data set sizes having a detrimental affect on performance as observed with the type-b results. Thus, it is advantageous to use as few zones as possible when organizing the domain of the CFD case so that the I/O requests can write more data at a time and provide better performance. In the most extreme cases, the parallel version actually performed worse than the serial version.

Figure (5.11) shows the results of the type-d benchmarks of the pCGNS library, which test the library’s ability to provide multiple processes with access to the same data array in a file simultaneously. Unlike previous benchmarks, the rapid drop in bandwidth when switching from one to two nodes is not because of an I/O buffer advantage enjoyed by the serial job, but rather the library switching from a single process accessing the data array to multiple processes accessing the data array. While the HDF5 library which powers the parallel I/O capabilities of pCGNS supports this type of I/O, it obviously does not perform very well. This is likely due to the added complexity of dealing with possible overlaps in the data writes, since the library cannot assume that all ranges of data written will be independent from one another. As usual the bandwidth observed with larger data sizes is greater, but none of the runs achieved a speedup greater than one. These performance limitations do not invalidate the usefulness of pCGNS when operating in this mode. Even if the I/O performance may be worse than the serial case when processes share zones, the total possible problem size is still greater than allowed by serial I/O, since that requires that the entire problem fit into the RAM of a single node.

Like the type-d benchmarks, the type-e benchmarks also demonstrate poor performance, as shown by Figure (5.12). Again, these results can be explained by multiple processes accessing a single array in the file simultaneously. In type-e, each process also writes to multiple zones, so two features are multiply associated. This is the most general case explored by the benchmarks, and demonstrates the complexity present in CFD data when the physically significant zones in the
Fig. 5.8: Results of pCGNS benchmark (c). Two zones are written by each node while keeping the file size constant, causing less and less data to be written with each additional node. The performance is still acceptable, but does level off after four nodes.
Fig. 5.9: Results of pCGNS benchmark (c). Four zones are written by each node while keeping the file size constant, causing less and less data to be written with each additional node. The performance is no longer an improvement over the serial case with the smallest data size, although larger files still perform acceptably.
Fig. 5.10: Results of pCGNS benchmark (c). Eight zones are written by each node while keeping the file size constant, causing less and less data to be written with each additional node. The performance is no longer an improvement over the serial case with all but the largest data size, which barely performs better than the serial version.
Fig. 5.11: Results of pCGNS benchmark (d). Multiple nodes share each zone, requiring HDF5 to sort out any possible data overlaps. This causes a severe slow down of the I/O operations. None of the tested file sizes outperformed the serial case.
domain do not match up well with the computationally significant partitions. Also like the type-d results, these results are not totally negative, however, since the performance is not so bad that the method could not be considered as a means of easily writing very large case files which cannot be stored in the memory of any node on the cluster.

In general the results of benchmarking the pCGNS library are not unexpected. Larger file sizes enjoy better utilization of the parallel I/O hardware available, since more time is spent transferring data than coordinating the I/O operations between the processes. This effect is seen in nearly every benchmark run on the library, but is not truly problematic since the purpose of using a computing cluster when solving a CFD case is generally to handle larger domains than otherwise possible. Additionally, for performance reasons, it is critical that each process only access a single zone at a time. While this will reduce the total amount of data that can be transferred in each operation, the performance gains expected from this restriction far outweigh any reduction caused by smaller data sizes. In light of the poor performance when accessing a single zone from multiple processes, the queuing operations of the library should be altered in future versions to prevent this I/O mode from being used.

When considered against the small collective write benchmark done using IOR with the HDF5 interface, most of the pCGNS runs compare favorably. Even on the type-d and type-e benchmarks, the largest file sizes performed nearly as well as IOR in those same cases. This suggests that the pCGNS library successfully assumes the performance of the HDF5 library under these operating conditions.
Fig. 5.12: Results of pCGNS benchmark (e). The most general of the benchmarks, type-e demonstrates the performance when multiple zones are written by multiple processes. This situation results from parallel partitioning independent of the zone arrangement of the domain.
Chapter 6

Summary and Conclusion

The purpose of this work was to implement a new library which augments the CGNS Mid-level Library with parallel I/O write capabilities using the new features added in recent versions of the HDF5 library and benchmark the performance of the resulting software. This was done in hopes that the parallel extension would provide improved performance when compared with serial execution of the same code, but parallel I/O provides more benefits than just improved performance. Having parallel I/O capabilities simplifies the process of writing solutions to disk when running a simulation on a cluster computer, since there is no longer the need to manually synchronize the writes from each process or to write a single file from each process and recombine them after the simulation completes.

The decision made early in development to implement the new library directly using HDF5 instead of the ADF to HDF5 compatibility layer in the MLL not only made coding pCGNS much easier, but most likely made it possible at all. Fortunately, since CGNS supports both the ADF and HDF5 back ends in the main library and all the tools, there are no compatibility problems between files generated with pCGNS and the MLL.

By benchmarking the library developed for this work the performance of the library on the clusters at Utah State University using the Panasas storage system was measured and these results were compared against benchmarks run on the same hardware using only the HDF5 library through the IOR benchmarking tool. When run using multiple processors on a single node, the pCGNS library does experience variation in performance, but overall speedup of the code as processors are added is very near to one. Since the aggregate I/O bandwidth of all the processes on a node is limited by the network connection of that node, a speedup of one indicates that the library is operating efficiently. If the speedup were less than one, this would be due to processes coordination overhead incurred by running multiple jobs.
When running benchmarks in which multiple processes across multiple nodes wrote to separate array, the speedup of the code is nearly ideally linear for up to four nodes running a single process each. The maximum speedup was attained when using all nodes and performed four times better than the serial version. The limitation of the linear speedup to four nodes is likely a result of having only four shelves of Panasas storage.

The only cases for which the parallel version of the library did not outperform the serial version were those in which each array was accessed by multiple processes across the cluster. These cases experienced a speedup less than one, indicating that the code actually ran slower than the single process case. The results from these case do still compare favorably with the IOR benchmarks when executing the same operations, so the inefficiencies must be in the HDF5 library. In these cases, the parallel version of the code can still write more data from memory to disk than the serial version, since the domain is scattered across the nodes of the cluster.

It is hoped that this version of the pCGNS library will serve as the alpha release of the library for general adoption by the CGNS and CFD community. It will be presented at SC09 (Supercomputing '09) as Utah State University’s Storage Challenge submission. USU was selected as one of four finalists in this challenge, and faces competition from such organizations as NVIDIA, Microsoft, and IBM, as well as first tier universities. After presentation at the conference, the source code to the pCGNS library will be posted as a project on Sourceforge under an open source license so that those concerned may continue its development, as led by the CGNS steering committee. This decision is in line with the licensing and use of the rest of the CGNS software.
Bibliography


Appendix
Chapter A

pCGNS Source Code & Documentation

A.1 Data Structure Index

A.1.1 Data Structures

Here are the data structures with brief descriptions:

base_s (Struct to describe a base node in the file) .............................. 60
coords_s (Struct to describe a coords node in the file) ...................... 63
file_s (Struct to describe a CGNS file) ............................................ 64
iter_s (Structure to describe nodes during iteration) ......................... 69
section_s (Struct to describe a section node in the file) .................... 70
slice_s ................................................................. 72
sol_s (Struct to describe a solution node in the file) .......................... 74
zone_s (Struct to describe a zone node in the file) ............................ 76

A.2 File Index

A.2.1 File List

Here is a list of all files with brief descriptions:

benchmark.c ............................................................. 80
open_close.c ............................................................. 94
pcgns_util.c ............................................................ 97
pcgns_util.h ........................................................... 140
pcgnslib.c .............................................................. 168
pcgnslib.h .............................................................. 228
A.3 Data Structure Documentation

A.3.1 base_s Struct Reference

Struct to describe a base node in the file.

```c
#include <pcgns_util.h>
```

Collaboration diagram for base_s:

![Collaboration Diagram](image)

Data Fields

- `int idx`
  
  *Index of the node.*

- `char basename[100+1]`
  
  *Name of the node.*
• zone_t * zones

Array of zone_t objects which describe the zones in the base.

• int nzones

Length of the array [zones].

• hid_t group_id

HDF5 handle to the node.

• int cell_dim

Cell dimensions.

• int phys_dim

Physical dimensions.

A.3.1.1 Detailed Description

Struct to describe a base node in the file.

Definition at line 98 of file pcgns_util.h.

A.3.1.2 Field Documentation

A.3.1.2.1 char basename[100+1]

Name of the node.

Definition at line 102 of file pcgns_util.h.
A.3.1.2.2 int cell_dim

   Cell dimensions.
   Definition at line 110 of file pcgns_util.h.

A.3.1.2.3 hid_t group_id

   HDF5 handle to the node.
   Definition at line 108 of file pcgns_util.h.

A.3.1.2.4 int idx

   Index of the node.
   Definition at line 100 of file pcgns_util.h.

A.3.1.2.5 int nzones

   Length of the array {zones}.
   Definition at line 106 of file pcgns_util.h.

A.3.1.2.6 int phys_dim

   Physical dimensions.
   Definition at line 112 of file pcgns_util.h.

A.3.1.2.7 zone_t* zones

   Array of zone_t objects which describe the zones in the base.
Definition at line 104 of file pcgns_util.h.

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

- pcgns_util.h

A.3.2  coords_s Struct Reference

Struct to describe a coords node in the file.

#include <pcgns_util.h>

Data Fields

- int idx

  Index of the node.

- char coordname [100+1]

  Name of the node.

- hid_t group_id

  HDF5 handle to the node.

A.3.2.1  Detailed Description

Struct to describe a coords node in the file.

Definition at line 30 of file pcgns_util.h.

A.3.2.2  Field Documentation

A.3.2.2.1  char coordname[100+1]

  Name of the node.
A.3.2.2.2  hid_t group_id

HDF5 handle to the node.
Definition at line 36 of file pcgns_util.h.

A.3.2.2.3  int idx

Index of the node.
Definition at line 32 of file pcgns_util.h.
The documentation for this struct was generated from the following file:

* pcgns_util.h

A.3.3  file_s Struct Reference

Struct to describe a CGNS file.
#include <pcgns_util.h>

Collaboration diagram for file_s:

Data Fields

- int idx
  
  *The index of this file in {files}.*

- char filename [100+1]
  
  *The name of this file.*

- int isOpen
  
  *Flag to tell if this file is open.*

- base_t * bases
  
  *Array of base_t objects which describe the bases in the file.*
• int nbases

  Length of the array {bases}.

• hid_t file_id

  HDF5 handle to the file.

• hid_t plist_id

  HDF5 property list of the file.

• MPI_Comm comm

  MPI comm on which the file was opened.

• MPI_Info info

  MPI info.

• int rank

  MPI rank of this process in this comm.

• int size

  MPI size of this comm.

A.3.3.1 Detailed Description

  Struct to describe a CGNS file.

  Definition at line 116 of file pcgns_util.h.

A.3.3.2 Field Documentation
A.3.3.2.1 base_t* bases

    Array of base_t objects which describe the bases in the file.
    Definition at line 124 of file pcgns_util.h.

A.3.3.2.2 MPI_Comm comm

    MPI comm on which the file was opened.
    Definition at line 132 of file pcgns_util.h.

A.3.3.2.3 hid_t file_id

    HDF5 handle to the file.
    Definition at line 128 of file pcgns_util.h.

A.3.3.2.4 char filename[100+1]

    The name of this file.
    Definition at line 120 of file pcgns_util.h.

A.3.3.2.5 int idx

    The index of this file in {files}.
    Definition at line 118 of file pcgns_util.h.

A.3.3.2.6 MPI_Info info

    MPI info.
A.3.3.2.7  int isOpen

Flag to tell if this file is open.
Definition at line 122 of file pcgns_util.h.

A.3.3.2.8  int nbases

Length of the array {bases}.
Definition at line 126 of file pcgns_util.h.

A.3.3.2.9  hid_t plist_id

HDF5 property list of the file.
Definition at line 130 of file pcgns_util.h.

A.3.3.2.10  int rank

MPI rank of this process in this comm.
Definition at line 136 of file pcgns_util.h.

A.3.3.2.11  int size

MPI size of this comm.
Definition at line 138 of file pcgns_util.h.

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

• pcgns_util.h
A.3.4 iter_s Struct Reference

Structure to describe nodes during iteration.

Data Fields

• int pos
• int counter
• char * name
• char * label

A.3.4.1 Detailed Description

Structure to describe nodes during iteration.

Definition at line 522 of file pcgns_util.c.

A.3.4.2 Field Documentation

A.3.4.2.1 int counter

Definition at line 522 of file pcgns_util.c.

A.3.4.2.2 char * label

Definition at line 522 of file pcgns_util.c.

A.3.4.2.3 char * name

Definition at line 522 of file pcgns_util.c.
A.3.4.2.4  int pos

Definition at line 522 of file pcgns_util.c.
The documentation for this struct was generated from the following file:

• pcgns_util.c

A.3.5  section_s Struct Reference

Struct to describe a section node in the file.
#include <pcgns_util.h>

Data Fields

• int idx

  Index of the node.

• char sectionname [100+1]

  Name of the node.

• hid_t group_id

  HDF5 handle to the node.

• ElementType_t type

  Type of elements in the section.

• int range [2]

  Range of element ids in the section.

• hid_t connectivity_id
**HDF5 handle to ElementConnectivity.**

- hid_t range_id

  **HDF5 handle to ElementRange.**

### A.3.5.1 Detailed Description

Struct to describe a section node in the file.

Definition at line 40 of file pcgns_util.h.

### A.3.5.2 Field Documentation

#### A.3.5.2.1 hid_t connectivity_id

HDF5 handle to ElementConnectivity.

Definition at line 52 of file pcgns_util.h.

#### A.3.5.2.2 hid_t group_id

HDF5 handle to the node.

Definition at line 46 of file pcgns_util.h.

#### A.3.5.2.3 int idx

Index of the node.

Definition at line 42 of file pcgns_util.h.

#### A.3.5.2.4 int range[2]


Range of element ids in the section.
Definition at line 50 of file pcgns_util.h.

A.3.5.2.5 hid_t range_id

HDF5 handle to ElementRange.
Definition at line 54 of file pcgns_util.h.

A.3.5.2.6 char sectionname[100+1]

Name of the node.
Definition at line 44 of file pcgns_util.h.

A.3.5.2.7 ElementType_t type

Type of elements in the section.
Definition at line 48 of file pcgns_util.h.
The documentation for this struct was generated from the following file:

• pcgns_util.h

A.3.6 slice_s Struct Reference

#include <pcgns_util.h>

Data Fields

• SliceType_t type
• int rank
• int * min
• int * max
• void * data
• int F
• int B
• int Z
• int Selector
• char name [100+1]

A.3.6.1 Detailed Description

Definition at line 141 of file pcgns_util.h.

A.3.6.2 Field Documentation

A.3.6.2.1 int B

Definition at line 148 of file pcgns_util.h.

A.3.6.2.2 void* data

Definition at line 146 of file pcgns_util.h.

A.3.6.2.3 int F

Definition at line 147 of file pcgns_util.h.

A.3.6.2.4 int* max

Definition at line 145 of file pcgns_util.h.
A.3.6.2.5 int* min

Definition at line 144 of file pcgns_util.h.

A.3.6.2.6 char name[100+1]

Definition at line 151 of file pcgns_util.h.

A.3.6.2.7 int rank

Definition at line 143 of file pcgns_util.h.

A.3.6.2.8 int Selector

Definition at line 150 of file pcgns_util.h.

A.3.6.2.9 SliceType_t type

Definition at line 142 of file pcgns_util.h.

A.3.6.2.10 int Z

Definition at line 149 of file pcgns_util.h.

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

- pcgns_util.h

A.3.7 sol_s Struct Reference

Struct to describe a solution node in the file.
#include <pcgns_util.h>

Data Fields

- int idx
  
  *Index of the node.*

- char solname [100+1]
  
  *Name of the node.*

- hid_t group_id
  
  *HDF5 handle to the node.*

A.3.7.1 Detailed Description

Struct to describe a solution node in the file.

Definition at line 58 of file pcgns_util.h.

A.3.7.2 Field Documentation

A.3.7.2.1 hid_t group_id

HDF5 handle to the node.

Definition at line 64 of file pcgns_util.h.

A.3.7.2.2 int idx

Index of the node.

Definition at line 60 of file pcgns_util.h.
A.3.7.2.3 char solname[100+1]

Name of the node.
Definition at line 62 of file pcgns_util.h.
The documentation for this struct was generated from the following file:

• pcgns_util.h

A.3.8 zone_s Struct Reference

Struct to describe a zone node in the file.
#include <pcgns_util.h>Collaboration diagram for zone_s:

Data Fields

• int idx

    Index of the node.

• char zonename [100+1]

    Name of the node.

• coords_t *coords

    Array of coord_t objects which describe the coords in the zone.
• int ncoords

    Length of the array {coords}.

• section_t * sections

    Array of section_s objects which describe the element sections in the zone.

• int nsections

    Length of the array {sections}.

• sol_t * sols

    Array of sol_t objects which describe the solutions in the zone.

• int nsols

    Length of the array {sols}.

• hid_t group_id

    HDF5 handle to the node.

• int * nijk

    Zone dimensions.

• ZoneType_t type

    Zone type.

• hid_t grid_id

    HDF5 handle to the grid node.

• hid_t flow_id
A.3.8.1 Detailed Description

Struct to describe a zone node in the file.
Definition at line 68 of file pcgns_util.h.

A.3.8.2 Field Documentation

A.3.8.2.1 coords_t* coords

Array of coord_t objects which describe the coords in the zone.
Definition at line 74 of file pcgns_util.h.

A.3.8.2.2 hid_t flow_id

HDF5 handle to the flow node.
Definition at line 94 of file pcgns_util.h.

A.3.8.2.3 hid_t grid_id

HDF5 handle to the grid node.
Definition at line 92 of file pcgns_util.h.

A.3.8.2.4 hid_t group_id

HDF5 handle to the node.
Definition at line 86 of file pcgns_util.h.
A.3.8.2.5  int idx

Index of the node.
Definition at line 70 of file pcgns_util.h.

A.3.8.2.6  int ncoords

Length of the array {coords}.
Definition at line 76 of file pcgns_util.h.

A.3.8.2.7  int* nijk

Zone dimensions.
Definition at line 88 of file pcgns_util.h.

A.3.8.2.8  int nsections

Length of the array {sections}.
Definition at line 80 of file pcgns_util.h.

A.3.8.2.9  int nsols

Length of the array {sols}.
Definition at line 84 of file pcgns_util.h.

A.3.8.2.10  section_t* sections

Array of section_s objects which describe the element sections in the zone.
Definition at line 78 of file pcgns_util.h.

A.3.8.2.11  sol_t* sols

Array of sol_t objects which describe the solutions in the zone.
Definition at line 82 of file pcgns_util.h.

A.3.8.2.12  ZoneType_t type

Zone type.
Definition at line 90 of file pcgns_util.h.

A.3.8.2.13  char zonename[100+1]

Name of the node.
Definition at line 72 of file pcgns_util.h.

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

•  pcgns_util.h

A.4  File Documentation

A.4.1  benchmark.c File Reference

#include "pcgnslib.h"
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"
#include "string.h"
#include "math.h"
Include dependency graph for benchmark.c:

![Dependency Graph](image)

Defines

- `#define MEGA_BYTES 256`
- `#define BUF_LENGTH (MEGA_BYTES∗1024∗1024/sizeof(double))`
- `#define N ((int) sqrt((double) BUF_LENGTH))`

Functions

- `int initialize (int *argc, char **argv[])`
- `int finalize ()`
- `int doTimer (const char *msg, double time)`
- `int doBandwidth (const char *msg, double time)`
- `int doBandwidthAgg (const char *msg, double time)`
- `int main (int argc, char *argv[])`

Variables

- `int comm_size`
- `int comm_rank`
- `MPI_Info info`
- `int nijk [3][3]`
- `double * x`
- `double * y`
A.4.1.1 Detailed Description

Author:

Kyle Horne <horne.kyle@gmail.com>

Version:

0.2

A.4.1.2 LICENSE

BSD style license

A.4.1.3 DESCRIPTION

Test program for pcgns library

Definition in file benchmark.c.

A.4.1.4 Define Documentation

A.4.1.4.1 #define BUF_LENGTH (MEGA_BYTES * 1024 * 1024 / sizeof(double))

• double * z
• int min [3]
• int max [3]
• int fn
• int B
• int Z
• int C
• double t0
• double t1
• double ta
A.4.1.4.2 #define MEGA_BYTES 256

Definition at line 20 of file benchmark.c.

A.4.1.4.3 #define N ((int) sqrt((double) BUF_LENGTH))

Definition at line 19 of file benchmark.c.

A.4.1.5 Function Documentation

A.4.1.5.1 int doBandwidth (const char *msg, double time)

Definition at line 108 of file benchmark.c.
Here is the caller graph for this function:

A.4.1.5.2 int doBandwidthAgg (const char *msg, double time)

Definition at line 125 of file benchmark.c.
Here is the caller graph for this function:

A.4.1.5.3 int doTimer (const char *msg, double time)
Definition at line 96 of file benchmark.c.
Here is the caller graph for this function:

A.4.1.5.4 int finalize (void)

Definition at line 86 of file benchmark.c.
Here is the caller graph for this function:

A.4.1.5.5 int initialize (int *argc, char **argv[])

Definition at line 46 of file benchmark.c.
Here is the caller graph for this function:

A.4.1.5.6 int main (int argc, char *argv[])

Definition at line 142 of file benchmark.c.
Here is the call graph for this function:

A.4.1.6 Variable Documentation

A.4.1.6.1 int B

Definition at line 38 of file benchmark.c.

A.4.1.6.2 int C

Definition at line 40 of file benchmark.c.
A.4.1.6.3  int comm_rank

Definition at line 26 of file benchmark.c.

A.4.1.6.4  int comm_size

Definition at line 25 of file benchmark.c.

A.4.1.6.5  int fn

Definition at line 37 of file benchmark.c.

A.4.1.6.6  MPI_Info info

Definition at line 27 of file benchmark.c.

A.4.1.6.7  int max[3]

Definition at line 35 of file benchmark.c.

A.4.1.6.8  int min[3]

Definition at line 34 of file benchmark.c.

A.4.1.6.9  int nijk[3][3]

Definition at line 28 of file benchmark.c.
A.4.1.6.10  double t0

Definition at line 42 of file benchmark.c.

A.4.1.6.11  double t1

Definition at line 43 of file benchmark.c.

A.4.1.6.12  double ta

Definition at line 44 of file benchmark.c.

A.4.1.6.13  double* x

Definition at line 30 of file benchmark.c.

A.4.1.6.14  double* y

Definition at line 31 of file benchmark.c.

A.4.1.6.15  int Z

Definition at line 39 of file benchmark.c.

A.4.1.6.16  double* z

Definition at line 32 of file benchmark.c.
A.4.2 benchmark.c

```c
#include "pcgnslib.h"
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"
#include "string.h"
#include "math.h"

#define MEGA_BYTES 256
#define BUF_LENGTH (MEGA_BYTES*1024*1024/sizeof(double))

#define N ((int) sqrt((double) BUF_LENGTH))

int comm_size;
int comm_rank;
MPI_Info info;
int nijk[3][3];

double* x;
double* y;
double* z;
int min[3];
```
int max[3];
int fn;
int B;
int Z;
int C;

double t0;
double t1;
double ta;

int initialize(int* argc, char** argv[]) {
    MPI_Init(argc,argv);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &comm_rank);
    MPI_Info_create(&info);

    nijk[0][0] = N;
    nijk[0][1] = N;
    nijk[0][2] = comm_size;
    nijk[1][0] = nijk[0][0]-1;
    nijk[1][1] = nijk[0][1]-1;
    nijk[1][2] = nijk[0][2]-1;
    nijk[2][0] = 0;
    nijk[2][1] = 0;
    nijk[2][2] = 0;

    x = (double*) malloc(BUF_LENGTH*sizeof(double));
    y = (double*) malloc(BUF_LENGTH*sizeof(double));
    z = (double*) malloc(BUF_LENGTH*sizeof(double));

    for(i=0; i<N; i++) {
        for(j=0; j<N; j++) {
            x[i*N+j] = (double) (i);
y[i*N+j] = (double) (j);

z[i*N+j] = (double) (comm_rank);

}

min[0] = comm_rank;
min[1] = 0;
min[2] = 0;
max[0] = comm_rank;
max[1] = N-1;
max[2] = N-1;

return 0;

int finalize() {
free(x);
free(y);
free(z);

MPI_Finalize();

return 0;
}

int doTimer(const char* msg, double time) {
  double min;
double max;
double avg;

MPI_Reduce(&time, &min, 1, MPI_DOUBLE, MPI_MIN, 0, MPI_COMM_WORLD);
MPI_Reduce(&time, &max, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
MPI_Reduce(&time, &avg, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

avg = avg/((double) comm_size);

if(comm_rank==0) printf("%20s Time = { min: %-20f max: %-20f avg: %-20f} s\n", msg, min, max, avg);
msg, min, max, avg);

int doBandwidth(const char* msg, double time) {
    double min;
    double max;
    double avg;

double MB = ((double) BUF_LENGTH * sizeof(double))/ (1024.0 * 1024.0);
MPI_Reduce(&time, &max, 1, MPI_DOUBLE, MPI_MIN, 0, MPI_COMM_WORLD);
max = MB/max;
MPI_Reduce(&time, &min, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
min = MB/min;
MPI_Reduce(&time, &avg, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
avg = avg/((double) comm_size);
    if (comm_rank==0) printf("%s Band = { min: %-20f max: %-20f avg: %-20f} MB/s (local)\n", msg, min, max, avg);
    return 0;
}

int doBandwidthAgg(const char* msg, double time) {
    double min;
    double max;
    double avg;
    double MB = ((double) BUF_LENGTH * sizeof(double))/ (1024.0 * 1024.0) * ((double) comm_size);
MPI_Reduce(&time, &max, 1, MPI_DOUBLE, MPI_MIN, 0, MPI_COMM_WORLD);
max = MB/max;
MPI_Reduce(&time, &min, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
min = MB/min;
MPI_Reduce(&time, &avg, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
avg = avg/((double) comm_size);
```c
00137    avg = MB/avg;
00138  if(comm_rank==0) printf("%20s Band = { min: %-20f max: %-20f avg: %-20f} MB/s
00139                   (aggregate)\n",msg,min,max,avg);
00140    return 0;
00141 }
00142
00143 int main(int argc, char* argv[]) {  
00144    // Initialize variables
00145    initialize(&argc,&argv);
00146  
00147     // Time the creation of a file
00148     t0 = MPI_Wtime();
00149     cgp_open("benchmark.cgns", 0, MPI_COMM_WORLD, &info, &fn);
00150     t1 = MPI_Wtime();
00151     doTimer("File Open", t1-t0);
00152  
00153     // Time the creation of a base
00154     t0 = MPI_Wtime();
00155     cgp_base_write(fn, "Base 1", 3, 3, &B);
00156     t1 = MPI_Wtime();
00157     doTimer("Base Write", t1-t0);
00158  
00159     // Time the creation of a zone
00160     t0 = MPI_Wtime();
00161     cgp_zone_write(fn, B, "Zone 1", &(nijk[0][0]), 0, &Z);
00162     t1 = MPI_Wtime();
00163     doTimer("Zone Write", t1-t0);
00164  
00165     // Time the creation of coordinates X
00166     t0 = MPI_Wtime();
00167     cgp_coord_write(fn,B,Z,0,"CoordinateX",&C);
00168     t1 = MPI_Wtime();
00169     doTimer("Coord X Write", t1-t0);
00170  
00171     // Time the write speed of coordinates X
```
MPI_Barrier(MPI_COMM_WORLD);
t0 = MPI_Wtime();
cgp_coord_write_data(fn,B,Z,C,min,max,x);
t1 = MPI_Wtime();
MPI_Barrier(MPI_COMM_WORLD);
ta = MPI_Wtime();
doTimer("Coord X Write Data", t1-t0);
doBandwidth("Coord X Write Data", t1-t0);
doBandwidthAgg("Coord X Write Data", ta-t0);

// Time the creation of coordinates Y
t0 = MPI_Wtime();
cgp_coord_write(fn,B,Z,0,"CoordinateY",&C);
t1 = MPI_Wtime();
doTimer("Coord Y Write", t1-t0);

// Time the write speed of coordinates Y
MPI_Barrier(MPI_COMM_WORLD);
t0 = MPI_Wtime();
cgp_coord_write_data(fn,B,Z,C,min,max,y);
t1 = MPI_Wtime();
MPI_Barrier(MPI_COMM_WORLD);
ta = MPI_Wtime();
doTimer("Coord Y Write Data", t1-t0);
doBandwidth("Coord Y Write Data", t1-t0);
doBandwidthAgg("Coord Y Write Data", ta-t0);

// Time the creation of coordinates Z
t0 = MPI_Wtime();
cgp_coord_write(fn,B,Z,0,"CoordinateZ",&C);
t1 = MPI_Wtime();
doTimer("Coord Z Write", t1-t0);
d0205
// Time the write speed of coordinates Z
MPI_Barrier(MPI_COMM_WORLD);
t0 = MPI_Wtime();
cgp_coord_write_data(fn,B,Z,C,min,max,z);
t1 = MPI_Wtime();
MPI_Barrier(MPI_COMM_WORLD);
ta = MPI_Wtime();

doTimer("Coord Z Write Data", t1-t0);
doSspeed("Coord Z Write Data", t1-t0);
doBandwidthAgg("Coord Z Write Data", ta-t0);

// Time closing of the file
t0 = MPI_Wtime();
cgp_close(fn);
t1 = MPI_Wtime();
doTimer("File Close", t1-t0);

finalize();

return 0;
}

A.4.3 open_close.c File Reference

#include "pcgnslib.h"
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"
Include dependency graph for open_close.c:

```
open_close.c
  |   |
  v   v
pcgnslib.h  stdio.h  stdlib.h
  |   |
  v   v
mpi.h  hdf5.h
```

Functions

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

A.4.3.1 Detailed Description

Author:

Kyle Horne <horne.kyle@gmail.com>

Version:

0.2

A.4.3.2 LICENSE

BSD style license

A.4.3.3 DESCRIPTION

Test program for pcgns library

Definition in file open_close.c.

A.4.3.4 Function Documentation

A.4.3.4.1 int main (int argc, char *argv[ ])

A.4.3.4.1 int main (int argc, char *argv[ ])

Definition at line 17 of file open_close.c.

Here is the call graph for this function:

```
A.4.4 open_close.c

00001
00002
00003
00004
00005
00006
00007
00008
00009
00010
00011 #include "pcgnslib.h"
00012
00013 #include "stdio.h"
00014 #include "stdlib.h"
00015 #include "mpi.h"
```
```c
int main(int argc, char* argv[]) {
    int err;
    int comm_size;
    int comm_rank;
    MPI_Info info;
    int fn;
    err = MPI_Init(&argc,&argv);
    if(err!=MPI_SUCCESS) cgp_doError;
    err = MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    if(err!=MPI_SUCCESS) cgp_doError;
    err = MPI_Comm_rank(MPI_COMM_WORLD, &comm_rank);
    if(err!=MPI_SUCCESS) cgp_doError;
    err = MPI_Info_create(&(info));
    if(err!=MPI_SUCCESS) cgp_doError;
    err = cgp_open("open_close.cgns", 0, MPI_COMM_WORLD, &info, &fn);
    if(err!=0) cgp_doError;
    err = cgp_close(fn);
    if(err!=0) cgp_doError;
    err = MPI_Finalize();
    if(err!=MPI_SUCCESS) cgp_doError;
    return err;
}
```

**A.4.5 pcgns_util.c File Reference**

```c
#include "pcgns_util.h"
#include "stdlib.h"
#include "string.h"
```
Include dependency graph for pcgns_util.c:

Data Structures

- struct iter_s
  
  *Structure to describe nodes during iteration.*

Functions

- void cleanup_files (void)
  
  *Function to free the array {files} at exit.*

- int next_file (int *fn)
- int free_file (file_t *file)
- int free_base (base_t *base)
- int free_zone (zone_t *zone)
- int free_coord (coords_t *coords)
- int free_section (section_t *section)
- int free_sol (sol_t *sol)
- int new_str (hid_t pid, const char *name, const char *value, int len)
- int get_str (hid_t pid, const char *name, int len, char *value)
- int new_str_attb (hid_t pid, const char *name, const char *value, int len)
• int get_str_attb (hid_t pid, const char *name, int len, char *value)
• int new_int (hid_t pid, const char *name, const int *value)
• int get_int (hid_t pid, const char *name, int *value)
• int new_int_attb (hid_t pid, const char *name, const int *value)
• int get_int_attb (hid_t pid, const char *name, int *value)
• int new_float (hid_t pid, const char *name, const float *value)
• int get_float (hid_t pid, const char *name, float *value)
• int new_float_attb (hid_t pid, const char *name, const float *value)
• int get_float_attb (hid_t pid, const char *name, float *value)
• int new_node (hid_t pid, const char *name, const char *label, const char *type)
• int del_node (hid_t pid, const char *name)
• int node_exists (hid_t pid, const char *name)
• herr_t node_counter (hid_t gid, const char *name, const H5L_info_t *linfo, void *vdata)
   
   Call-back function used to count the nodes with a particular label.

• int num_nodes (hid_t pid, const char *label, int *num)
• herr_t node_idx_finder (hid_t gid, const char *name, const H5L_info_t *linfo, void *vdata)
   
   Call-back function used to find the index of a nodes with a particular label.

• int node_name2idx (hid_t pid, const char *label, const char *name, int *idx)
• herr_t node_name_finder (hid_t gid, const char *name, const H5L_info_t *linfo, void *vdata)
   
   Call-back function used to find the name of a node at index [idx].

• int node_idx2name (hid_t pid, const char *label, int idx, char *name)
• int hdf5_version_str (int len, char *buf)
• int hdf5_format_str (int len, char *buf)
Variables

- $\text{file}_t \ast \text{files} = \text{NULL}$
  
  *Array of file_t’s which describe the open files.*

- $\text{int files}_\text{count} = 0$
  
  *Internal count of used slots in {files}.*

- $\text{int files}_\text{size} = 0$
  
  *Internal count of the slots in {files}.*

- $\text{slice}_t \ast \text{write}_\text{queue} = \text{NULL}$
  
  *Queue of IO write operations.*

- $\text{int write}_\text{queue}_\text{len} = 0$
  
  *Length of write queue.*

A.4.5.1 Detailed Description

Author:

Kyle Horne <horne.kyle@gmail.com>

Version:

0.2

A.4.5.2 LICENSE

BSD style license

A.4.5.3 DESCRIPTION
Implementation of utility functions
Definition in file pcgns_util.c.

A.4.5.4 Function Documentation

A.4.5.4.1 void cleanup_files (void)

Function to free the array {files} at exit.
Definition at line 33 of file pcgns_util.c.
Here is the caller graph for this function:

A.4.5.4.2 int del_node (hid_t pid, const char * name)

Delete a node with parent {pid}

Parameters:

  pid [in]: HDF5 locator for parent

  name [in]: Name of node

Returns:

Error code

Definition at line 506 of file pcgns_util.c.
Here is the caller graph for this function:

![Caller Graph](image)

### A.4.5.4.3 int free_base (base_t * base)

Free the memory of a base_t object

**Parameters:**

- **base** [in]: Pointer the base

**Returns:**

Error code

Definition at line 103 of file pcgns_util.c.

Here is the call graph for this function:

![Call Graph](image)

Here is the caller graph for this function:
A.4.5.4.4 int free_coord (coords_t * coords)

Definition at line 153 of file pcgns_util.c.

Here is the caller graph for this function:

A.4.5.4.5 int free_file (file_t * file)

Free the memory of a file_t object

Parameters:

file [in]: Pointer the file

Returns:

Error code

Definition at line 69 of file pcgns_util.c.

Here is the call graph for this function:

Here is the caller graph for this function:

A.4.5.4.6 int free_section (section_t * section)

Free the memory of a section_t object
A.4.5.4.7 int free_sol(sol_t *sol)

Free the memory of a sol_t object

Parameters:

   sol [in]: Pointer the sol

Returns:

   Error code

Definition at line 179 of file pcgns_util.c.

Here is the caller graph for this function:

A.4.5.4.8 int free_zone(zone_t *zone)

Free the memory of a zone_t object

Parameters:

   zone [in]: Pointer the zone

Returns:

   Error code

Definition at line 163 of file pcgns_util.c.

Here is the caller graph for this function:
Parameters:

zone [in]: Pointer the zone

Returns:

Error code

Definition at line 121 of file pcgns_util.c.

Here is the call graph for this function:

Here is the caller graph for this function:

A.4.5.4.9 int get_float (hid_t pid, const char ∗ name, float ∗ value)

Read an float with parent {pid}, name {name}, and value {value}

Parameters:

pid [in]: HDF5 locator for parent

name [in]: Name of float

value [out]: Float to read

Returns:

Error code

Definition at line 412 of file pcgns_util.c.
A.4.5.4.10 int get_float_attb (hid_t pid, const char * name, float * value)

Read an float attribute at location {pid} with {name} with {value}

Parameters:

   pid [in]: HDF5 locator for parent

   name [in]: Name of float

   value [out]: Float to read

Returns:

   Error code

   Definition at line 456 of file pcgns_util.c.

A.4.5.4.11 int get_int (hid_t pid, const char * name, int * value)

Read an integer with parent {pid}, name {name}, and value {value}

Parameters:

   pid [in]: HDF5 locator for parent

   name [in]: Name of integer

   value [out]: Integer to read

Returns:

   Error code

   Definition at line 322 of file pcgns_util.c.

A.4.5.4.12 int get_int_attb (hid_t pid, const char * name, int * value)

Read an integer attribute at location {pid} with {name} with {value}

Parameters:

   pid [in]: HDF5 locator for parent
name [in]: Name of integer

value [out]: Integer to read

Returns:

Error code

Definition at line 366 of file pcgns_util.c.

A.4.5.4.13 int get_str (hid_t pid, const char *name, int len, char *value)

Read a string with parent {pid}, name {name}, and value {value}

Parameters:

pid [in]: HDF5 locator for parent

name [in]: Name of string

len [in]: Length of string in file

value [out]: Contents of string

Returns:

Error code

Definition at line 214 of file pcgns_util.c.

Here is the caller graph for this function:

A.4.5.4.14 int get_str_attb (hid_t pid, const char *name, int len, char *value)

Read a string attribute at location {pid} with {name} with {value}

Parameters:

pid [in]: HDF5 locator for parent

name [in]: Name of string
**len** [in]: Length of string in file

**value** [out]: Contents of string

**Returns:**

Error code

Definition at line 269 of file p cgns_util.c.

Here is the caller graph for this function:

A.4.5.4.15  int hdf5_format_str (int *len, char *buf)

Fill a buffer with the HDF5 format string

**Parameters:**

**len** [in]: Length of buf

**buf** [out]: String with the HDF5 format

**Returns:**

Error code

Definition at line 706 of file p cgns_util.c.

Here is the caller graph for this function:

A.4.5.4.16  int hdf5_version_str (int *len, char *buf)

Fill a buffer with the HDF5 version number
Parameters:

*len*  [in]: Length of *buf*

*buf*  [out]: String with HDF5 version number

Returns:

Error code

Definition at line 692 of file pcgns_util.c.

Here is the caller graph for this function:

```
  hdf5_version_str --> cgp_open --> main
```

A.4.5.4.17  **int new_float (hid_t *pid, const char *name, const float *value)**

Create a new float with parent *{pid}* name *{name}* and value *{value}*

Parameters:

*pid*  [in]: HDF5 locator for parent

*name*  [in]: Name of float

*value*  [in]: Float to write

Returns:

Error code

Definition at line 390 of file pcgns_util.c.

Here is the caller graph for this function:

```
  new_float --> cgp_open --> main
```

A.4.5.4.18  **int new_float_attb (hid_t *pid, const char *name, const float *value)**

Create a new float attribute at location *{pid}* with *{name}* with *{value}*
Parameters:

`pid` [in]: HDF5 locator for parent

`name` [in]: Name of float

`value` [in]: Float to write

Returns:

Error code

Definition at line 434 of file pcgns_util.c.

A.4.5.4.19 int new_int (hid_t pid, const char *name, const int *value)

Create a new integer with parent {pid}, name {name}, and value {value}

Parameters:

`pid` [in]: HDF5 locator for parent

`name` [in]: Name of integer

`value` [in]: Integer to write

Returns:

Error code

Definition at line 300 of file pcgns_util.c.

Here is the caller graph for this function:

A.4.5.4.20 int new_int_attb (hid_t pid, const char *name, const int *value)

Create a new integer attribute at location {pid} with {name} with {value}

Parameters:

`pid` [in]: HDF5 locator for parent
**name** [in]: Name of integer

**value** [in]: Integer to write

**Returns:**

Error code

Definition at line 344 of file pcgns_util.c.

Here is the caller graph for this function:

```
A.4.5.4.21  int new_node (hid_t pid, const char * name, const char * label, const char * type)
```

Create a new node at with parent \{pid\}

**Parameters:**

- **pid** [in]: HDF5 locator for parent
- **name** [in]: Name of node
- **label** [in]: Label of node
- **type** [in]: Type of node

**Returns:**

Error code
A.4.5.4.22   int new_str (hid_t pid, const char *name, const char *value, int len)

Create a new string with parent {pid}, name {name}, and value {value}

Parameters:

pid  [in]: HDF5 locator for parent

name  [in]: Name of string

value  [in]: Contents of string

len  [in]: Length of string in file

Returns:

Error code

Definition at line 192 of file pcgns_util.c.
Here is the caller graph for this function:

```
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>new_str</td>
<td>new_str</td>
</tr>
<tr>
<td>cgp_open</td>
<td>cgp_zone_write</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------</td>
</tr>
</tbody>
</table>
```

A.4.5.4.23  `int new_str_attb (hid_t pid, const char * name, const char * value, int len)`

Create a new string attribute at location {pid} with {name} with {value}

**Parameters:**

- **pid** [in]: HDF5 locator for parent
- **name** [in]: Name of string
- **value** [in]: Contents of string
- **len** [in]: Length of string in file

**Returns:**

Error code

Definition at line 239 of file pcgns_util.c.

Here is the caller graph for this function:

```
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>new_str_attb</td>
<td>new_node</td>
</tr>
<tr>
<td>cgp_open</td>
<td>cgp_base_write</td>
</tr>
<tr>
<td>new_str_attb</td>
<td>cgp_coord_write</td>
</tr>
<tr>
<td>cgp_open</td>
<td>cgp_array_write</td>
</tr>
<tr>
<td>cgp_open</td>
<td>cgp_section_write</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------</td>
</tr>
</tbody>
</table>
```

```
A.4.5.4.24 int next_file (int * fn)

Return the number of the next available file_t in {files}

Parameters:

fn [out]: Index of next file

Returns:

Error code

Definition at line 37 of file pcgns_util.c.

Here is the call graph for this function:

Here is the caller graph for this function:

A.4.5.4.25 herr_t node_counter (hid_t gid, const char * name, const H5L_info_t * linfo, void * vdata)

Call-back function used to count the nodes with a particular label.

Definition at line 525 of file pcgns_util.c.

Here is the call graph for this function:

Here is the caller graph for this function:
A.4.5.4.26  int node_exists (hid_t pid, const char * name)

Check if a node exists with name {name} and parent {pid}

Parameters:

  pid [in]: HDF5 locator for parent

  name [in]: Name of node

Returns:

{1=Exists, 0=Does not exist}

Definition at line 514 of file pcgns_util.c.

Here is the caller graph for this function:

A.4.5.4.27  int node_idx2name (hid_t pid, const char * label, int idx, char * name)

Convert a node id to a node name

Parameters:

  pid [in]: HDF5 locator for parent

  label [in]: Label of node

  idx [in]: Index of node

  name [out]: Name of node
Definition at line 672 of file pcgns_util.c.

Here is the call graph for this function:

```
node_idx2name -> node_name_finder -> get_str_attr
```

Here is the caller graph for this function:

```
node_idx2name -> cgz_base_read -> main
node_idx2name -> cgz_open -> main
cgp_zone_read -> main
```

A.4.5.4.28  **herr_t node_idx_finder (hid_t gid, const char * name, const H5L_info_t * linfo, void * vdata)**

Call-back function used to find the index of a node with a particular label.

Definition at line 573 of file pcgns_util.c.

Here is the call graph for this function:

```
node_idx_finder -> get_str_attr
```

Here is the caller graph for this function:

```
node_idx_finder -> node_name2idx
```

A.4.5.4.29  **int node_name2idx (hid_t pid, const char * label, const char * name, int * idx)**

Convert a node name to a node id

**Parameters:**

*pid*  [in]: HDF5 locator for parent

*label*  [in]: Label of node
**name** [in]: Name of node

**idx** [out]: Index of node

**Returns:**

Error code

Definition at line 612 of file pcgns_util.c.

Here is the call graph for this function:

```
node_name2idx -> node_idx_finder -> get_str_attb
```

A.4.5.4.30 **herr_t node_name_finder (hid_t gid, const char * name, const H5L_info_t * linfo, void * vdata)**

Call-back function used to find the name of a node at index {idx}.

Definition at line 633 of file pcgns_util.c.

Here is the call graph for this function:

```
node_name_finder -> get_str_attb
```

Here is the caller graph for this function:

```
node_name_finder <- node_idx2name <- get_str_attb

node_name_finder <- node_idx2name <- cgp_base_read <- main
node_name_finder <- node_idx2name <- cgp_open <- main
node_name_finder <- node_idx2name <- cgp_zone_read <- main
```

A.4.5.4.31 **int num_nodes (hid_t pid, const char * label, int * num)**

Count the number of nodes of type {label} with parent {pid}

**Parameters:**

**pid** [in]: HDF5 locator for parent
label [in]: Label of nodes

num [out]: Number of nodes

Returns:

Error code

Definition at line 554 of file pcgns_util.c.

Here is the call graph for this function:

```
num_nodes -> node_counter -> get_str_atrb
```

Here is the caller graph for this function:

```
cgp_base_read -> main
num_nodes -> cgp_open -> main
cgp_zone_read -> main
```

A.4.5.5 Variable Documentation

A.4.5.5.1 file_t* files = NULL

Array of file_t's which describe the open files.

Definition at line 21 of file pcgns_util.c.

A.4.5.5.2 int files_count = 0

Internal count of used slots in {files}.

Definition at line 22 of file pcgns_util.c.
A.4.5.3  int files_size = 0

Internal count of the slots in {files}.
Definition at line 23 of file pcgns_util.c.

A.4.5.4  slice_t* write_queue = NULL

Queue of IO write operations.
Definition at line 25 of file pcgns_util.c.

A.4.5.5  int write_queue_len = 0

Length of write queue.
Definition at line 26 of file pcgns_util.c.

A.4.6  pcgns_util.c

00001
00002
00003
00004
00005
00006
00007
00008
00009
00010
00011  #include "pcgns_util.h"
00012
00013  #include "stdlib.h"
00014  #include "string.h"
00015
file_t* files = NULL;
int files_count = 0;
int files_size = 0;
slice_t* write_queue = NULL;
int write_queue_len = 0;

void cleanup_files(void) {
  if(files!=NULL) free(files);
}

int next_file(int* fn) {
  int err = 0;
  // Loop index
  int k;
  // Return the index of the next slot
  *fn = files_count;
  // Increment the number of used slots
  files_count++;
  // If this is the first execution, register to cleanup files at exit
  if(files_size==0) atexit(cleanup_files);
  // If the returned slot does not exist, extend the list so that it does
  if(files_size<=files_count) {
    // Keep track of how many slots were used
    int old_files_size = files_size;
    // Set the new size of the array

files_size = 2*files_count;

// Allocate a new array to replace the old one
file_t* new_files = malloc(2*files_count*sizeof(file_t));

if(new_files==NULL) cgp_doError;

// Copy the old array's data to the new array
for(k=0;k<old_files_size;k++) {
    new_files[k] = files[k];
}

// Free the old array
free(files);

// Point to the new array
files = new_files;

// Assume nothing bad happened
return 0;
}

int free_file(file_t* file) {
    printTime;
    int err = 0;
    herr_t herr;
    // Loop index
    int k;
    // Free all the bases in this file
    printTime;
    for(k=0;k<file->nbases;k++) err = free_base(&{file->bases[k]});
    printTime;
    if(err!=0) cgp_doError;
    // If bases was allocated, free it
    if(file->bases!=NULL) free(file->bases);
    // Free the MPI info object
    err = MPI_Info_free(&(file->info));
    if(err!=MPI_SUCCESS) cgp_doError;
    // Free the property list
    printTime;
herr = H5Pclose(file->plist_id);
printTime;
if(herr<0) cgp_doError;
// Free the HDF5 file
printTime;
herr = H5Fclose(file->file_id);
printTime;
if(herr<0) cgp_doError;
// Set the file’s status to closed
file->isOpen = FALSE;
// Zero out HDF5 pointers
file->plist_id = 0;
file->file_id = 0;
return 0;

int free_base(base_t* base) {
    int err = 0;
    herr_t herr;
    // Loop index
    int k;
    // Free the zones in this base
    for(k=0;k<base->nzones;k++) err = free_zone(&(base->zones[k]));
    if(err!=0) cgp_doError;
    // If zones was allocated, free it
    if(base->zones!=NULL) free(base->zones);
    // Free the HDF5 group
    herr = H5Gclose(base->group_id);
    if(herr<0) cgp_doError;
    // Zero out the HDF5 pointer
    base->group_id = 0;
    return 0;
}

int free_zone(zone_t* zone) {
int err = 0;
herr_t herr;

// Loop index
int k;

// Free the zone dimensions
if(zone->nijk!=NULL) free(zone->nijk);

// Free the coords in this zone
for(k=0;k<zone->ncoords;k++) err = free_coord(&(zone->coords[k]));
if(err!=0) cgp_doError;

// If coords was allocated, free it
if(zone->coords!=NULL) free(zone->coords);

// Free the sections in this zone
for(k=0;k<zone->nsections;k++) err = free_section(&(zone->sections[k]));
if(err!=0) cgp_doError;

// Free the sols in this zone
for(k=0;k<zone->nsols;k++) err = free_sol(&(zone->sols[k]));
if(err!=0) cgp_doError;

// If sols was allocated, free it
if(zone->sols!=NULL) free(zone->sols);

// Free the HDF5 groups
herr = H5Gclose(zone->group_id);
if(herr<0) cgp_doError;

herr = H5Gclose(zone->grid_id);
if(herr<0) cgp_doError;

herr = H5Gclose(zone->flow_id);
if(herr<0) cgp_doError;

// Zero out the HDF5 pointer
zone->group_id = 0;
return 0;

int free_coord(coords_t* coords) {
  herr_t herr;
  // Free the HDF5 group
  herr = H5Gclose(coords->group_id);
if(herr<0) cgp_doError;

// Zero out the HDF5 pointer
coords->group_id = 0;
return 0;
}

int free_section(section_t* section) {
  herr_t herr;
  // Free the HDF5 groups
  herr = H5Gclose(section->group_id);
  if(herr<0) cgp_doError;
  herr = H5Gclose(section->connectivity_id);
  if(herr<0) cgp_doError;
  herr = H5Gclose(section->range_id);
  if(herr<0) cgp_doError;
  // Zero out the HDF5 pointers
  section->group_id = 0;
  section->connectivity_id = 0;
  section->range_id = 0;
  return 0;
}

int free_sol(sol_t* sol) {
  herr_t herr;
  // Free the HDF5 group
  herr = H5Gclose(sol->group_id);
  if(herr<0) cgp_doError;
  // Zero out the HDF5 pointer
  sol->group_id = 0;
  return 0;
}

//==============================================================================
int new_str(hid_t pid, const char* name, const char* value, int len) {

define(herr_t herr);

    // Set the dimension of the data to the length of the string, including the terminator
    hsize_t dim = len + 1;

    // Create a shape object for the data
    hid_t shape_id = H5Screate_simple(1, &dim, NULL);
    if (shape_id < 0) cgp_doError;

    // Create the data in the file
    hid_t data_id = H5Dcreate2(pid, name, H5T_NATIVE_CHAR, shape_id, H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
    if (data_id < 0) cgp_doError;

    // Write the data to the file
    herr = H5Dwrite(data_id, H5T_NATIVE_CHAR, H5S_ALL, H5S_ALL, H5P_DEFAULT, value);
    if (herr < 0) cgp_doError;

    // Close the data
    herr = H5Dclose(data_id);
    if (herr < 0) cgp_doError;

    // Close the shape
    herr = H5Sclose(shape_id);
    if (herr < 0) cgp_doError;

    return 0;
}

int get_str(hid_t pid, const char* name, int len, char* value) {

define(herr_t herr);

    // Open the data in the file
    hid_t data_id = H5Dopen2(pid, name, H5P_DEFAULT);
    if (data_id < 0) cgp_doError;

    // Get the shape description
    hid_t shape_id = H5Dget_space(data_id);

    // Read the dimensions
    hsize_t dim;
    H5Sget_simple_extent_dims(shape_id, &dim, NULL);
if (dim>len) cgp_doError;

// Zero out the string
memset(value, '\0', len+1);

// Read the data from the file
herr = H5Dread(data_id, H5T_NATIVE_CHAR, shape_id, shape_id, H5P_DEFAULT, value);
if (herr<0) cgp_doError;

// Close the data
herr = H5Dclose(data_id);
if (herr<0) cgp_doError;

// Close the shape
herr = H5Sclose(shape_id);
if (herr<0) cgp_doError;
return 0;

int new_str_attb(hid_t pid, const char* name, const char* value, int len) {
    herr_t herr;
    // Create a shape object for the data
    hid_t shape_id = H5Screate(H5S_SCALAR);
    if (shape_id<0) cgp_doError;
    // Create a type for the data
    hid_t type_id = H5Tcopy(H5T_C_S1);
    if (type_id<0) cgp_doError;
    // Set the dimension of the data to the length of the string, including the terminator
    herr = H5Tset_size(type_id, len+1);
    if (herr<0) cgp_doError;
    // Create the data in the file
    hid_t attb_id = H5Acreate(pid, name, type_id, shape_id, H5P_DEFAULT, H5P_DEFAULT);
    if (attb_id<0) cgp_doError;
    // Write the data to the file
    herr = H5Awrite(attb_id, type_id, value);
    if (herr<0) cgp_doError;
// Close the data  
00257  herr = H5Aclose(attb_id); 
00258  if(herr<0) cgp_doError; 
00259  // Close the type  
00260  herr = H5Tclose(type_id); 
00261  if(herr<0) cgp_doError; 
00262  // Close the shape  
00263  herr = H5Sclose(shape_id); 
00264  if(herr<0) cgp_doError; 
00265  return 0;  
00266 }  
00267  
00268 // This probably needs to be fixed to be more like get_str() 
00269 int get_str_attb(hid_t pid, const char* name, int len, char* value) {  
00270  herr_t herr;  
00271  // Create a shape object for the data  
00272  hid_t shape_id = H5Screate(H5S_SCALAR);  
00273  if(shape_id<0) cgp_doError;  
00274  // Create a type for the data  
00275  hid_t type_id = H5Tcopy(H5T_C_S1);  
00276  if(type_id<0) cgp_doError;  
00277  // Set the dimension of the data to the length of the string, including the terminator  
00278  herr = H5Tset_size(type_id, len+1);  
00279  if(herr<0) cgp_doError;  
00280  // Open the data in the file  
00281  hid_t attb_id = H5Aopen(pid, name, H5P_DEFAULT);  
00282  if(attb_id<0) cgp_doError;  
00283  // Read the data from the file  
00284  herr = H5Aread(attb_id, type_id, value);  
00285  if(herr<0) cgp_doError;  
00286  // Close the data  
00287  herr = H5Aclose(attb_id);  
00288  if(herr<0) cgp_doError;  
00289  // Close the type
herr = H5Tclose(type_id);
if (herr<0) cgp_doError;
// Close the shape
herr = H5Sclose(shape_id);
if (herr<0) cgp_doError;
return 0;
}

//==============================================================================

int new_int(hid_t pid, const char* name, const int* value) {
    herr_t herr;
    hsize_t dim = 1;
    // Create a shape object for the data
    hid_t shape_id = H5Screate_simple(1,&dim, NULL);
    if (shape_id<0) cgp_doError;
    // Create the data in the file
    hid_t data_id = H5Dcreate2(pid, name, H5T_NATIVE_INT, shape_id, H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
    if (data_id<0) cgp_doError;
    // Write the data to the file
    herr = H5Dwrite(data_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT, value);
    if (herr<0) cgp_doError;
    // Close the data
    herr = H5Dclose(data_id);
    if (herr<0) cgp_doError;
    // Close the shape
    herr = H5Sclose(shape_id);
    if (herr<0) cgp_doError;
    return 0;
}

int get_int(hid_t pid, const char* name, int* value) {
herr_t herr;

// Set the dimension to be one element
hsize_t dim = 1;

// Create a shape object for the data
hid_t shape_id = H5Screate_simple(1, &dim, NULL);
if (shape_id < 0) cgp_doError;

// Open the data in the file
hid_t data_id = H5Dopen2(pid, name, H5P_DEFAULT);
if (data_id < 0) cgp_doError;

// Read the data from the file
herr = H5Dread(data_id, H5T_NATIVE_INT, shape_id, shape_id, H5P_DEFAULT, value);
if (herr < 0) cgp_doError;

// Close the data
herr = H5Dclose(data_id);
if (herr < 0) cgp_doError;

// Close the shape
herr = H5Sclose(shape_id);
if (herr < 0) cgp_doError;

return 0;

int new_int_attb(hid_t pid, const char* name, const int* value) {

herr_t herr;

// Set the dimension to be one element
hsize_t dim = 1;

// Create a shape object for the data
hid_t shape_id = H5Screate_simple(1, &dim, NULL);
if (shape_id < 0) cgp_doError;

// Create the data in the file
hid_t attb_id = H5Acreate(pid, name, H5T_NATIVE_INT, shape_id, shape_id, H5P_DEFAULT, H5P_DEFAULT);
if (attb_id < 0) cgp_doError;

// Write the data to the file
herr = H5Awrite(attb_id, H5T_NATIVE_INT, value);

}
if(herr<0) cgp_doError;

// Close the data
herr = H5Aclose(attb_id);
if(herr<0) cgp_doError;

// Close the shape
herr = H5Sclose(shape_id);
if(herr<0) cgp_doError;
return 0;

int get_int_attb(hid_t pid, const char* name, int* value) {
  herr_t herr;
  // Set the dimension to be one element
  hsize_t dim = 1;
  // Create a shape object for the data
  hid_t shape_id = H5Screate_simple(1,&dim, NULL);
  if(shape_id<0) cgp_doError;
  // Open the data in the file
  hid_t attb_id = H5Aopen(pid, name, H5P_DEFAULT);
  if(attb_id<0) cgp_doError;
  // Read the data from the file
  herr = H5Aread(attb_id, H5T_NATIVE_INT, value);
  if(herr<0) cgp_doError;
  // Close the data
  herr = H5Aclose(attb_id);
  if(herr<0) cgp_doError;
  // Close the shape
  herr = H5Sclose(shape_id);
  if(herr<0) cgp_doError;
  return 0;
}

int new_float(hid_t pid, const char* name, const float* value) {
herr_t herr;

// Set the dimension to be one element
hsize_t dim = 1;

// Create a shape object for the data
hid_t shape_id = H5Screate_simple(1, &dim, NULL);
if (shape_id < 0) cgp_doError;

// Create the data in the file
hid_t data_id = H5Dcreate2(pid, name, H5T_NATIVE_FLOAT, shape_id, H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
if (data_id < 0) cgp_doError;

// Write the data to the file
herr = H5Dwrite(data_id, H5T_NATIVE_FLOAT, H5S_ALL, H5S_ALL, H5P_DEFAULT, value);
if (herr < 0) cgp_doError;

// Close the data
herr = H5Dclose(data_id);
if (herr < 0) cgp_doError;

// Close the shape
herr = H5Sclose(shape_id);
if (herr < 0) cgp_doError;

return 0;

int get_float(hid_t pid, const char* name, float* value) {
    herr_t herr;
    // Set the dimension to be one element
    hsize_t dim = 1;
    // Create a shape object for the data
    hid_t shape_id = H5Screate_simple(1, &dim, NULL);
    if (shape_id < 0) cgp_doError;
    // Open the data in the file
    hid_t data_id = H5Dopen2(pid, name, H5P_DEFAULT);
    if (data_id < 0) cgp_doError;
    // Read the data from the file
    herr = H5Dread(data_id, H5T_NATIVE_FLOAT, shape_id, shape_id, H5P_DEFAULT, value);
    if (herr < 0) cgp_doError;
    return 0;
}
if (herr<0) cgp_doError;
// Close the data
herr = H5Dclose(data_id);
if (herr<0) cgp_doError;
// Close the shape
herr = H5Sclose(shape_id);
if (herr<0) cgp_doError;
return 0;

int new_float_attb(hid_t pid, const char* name, const float* value) {
    herr_t herr;
    // Set the dimension to be one element
    hsize_t dim = 1;
    // Create a shape object for the data
    hid_t shape_id = H5Screate_simple(1,&dim, NULL);
    if (shape_id<0) cgp_doError;
    // Create the data in the file
    hid_t attb_id = H5Acreate(pid, name, H5T_NATIVE_FLOAT, shape_id, H5P_DEFAULT, H5P_DEFAULT);
    if (attb_id<0) cgp_doError;
    // Write the data to the file
    herr = H5Awrite(attb_id, H5T_NATIVE_FLOAT, value);
    if (herr<0) cgp_doError;
    // Close the data
    herr = H5Aclose(attb_id);
    if (herr<0) cgp_doError;
    // Close the shape
    herr = H5Sclose(shape_id);
    if (herr<0) cgp_doError;
    return 0;
}

int get_float_attb(hid_t pid, const char* name, float* value) {
herr_t herr;

// Set the dimension to be one element
hsize_t dim = 1;

// Create a shape object for the data
hid_t shape_id = H5Screate_simple(1, &dim, NULL);
if (shape_id < 0) cgp_doError;

// Open the data in the file
hid_t attb_id = H5Aopen(pid, name, H5P_DEFAULT);
if (attb_id < 0) cgp_doError;

// Read the data from the file
herr = H5Aread(attb_id, H5T_NATIVE_FLOAT, value);
if (herr < 0) cgp_doError;

// Close the data
herr = H5Aclose(attb_id);
if (herr < 0) cgp_doError;

// Close the shape
herr = H5Sclose(shape_id);
if (herr < 0) cgp_doError;

return 0;

//==============================================================================

int new_node(hid_t pid, const char* name, const char* label, const char* type) {

int err = 0;

herr_t herr;

// Create the node in the file
hid_t group_id = H5Gcreate2(pid, name, H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
if (group_id < 0) cgp_doError;

// Set the label attribute; length is 32 for compatibility
err = new_str_attb(group_id, "label", label, 32);
if (err != 0) cgp_doError;

// Set the name attribute; length is 32 for compatibility
The name should always correspond to the HDF5 group name
err = new_str_attb(group_id, "name", name, 32);
if(err!=0) cgp_doError;
// Set the type attribute; length is 2 for compatibility
err = new_str_attb(group_id, "type", type, 2);
if(err!=0) cgp_doError;
// Set the flags attribute
int val = 1;
err = new_int_attb(group_id, "flags", &val);
if(err!=0) cgp_doError;
// Close the group
herr = H5Gclose(group_id);
if(herr<0) cgp_doError;
return 0;

int del_node(hid_t pid, const char* name) {
  herr_t herr;
  // Delete the node
  herr = H5Ldelete(pid, name, H5P_DEFAULT);
  if(herr<0) cgp_doError;
  return 0;
}

int node_exists(hid_t pid, const char* name) {
  // Check for the existence of a node
  return H5Lexists(pid, name, H5P_DEFAULT)?TRUE:FALSE;
}

//==============================================================================
struct iter_s {int pos; int counter; char* name; char* label;};

herr_t node_counter(hid_t gid, const char* name, const H5L_info_t* linfo, void* v
data) {
  int err = 0;
herr_t herr;
// Cast vdata as an iter_s
struct iter_s* data = (struct iter_s*) vdata;
// info object to hold data
H5O_info_t info;
// Get the info for the current object
herr = H5Oget_info_by_name(gid, name, &info, H5P_DEFAULT);
if(herr<0) cgp_doError;
// If the object is a group
if(info.type==H5O_TYPE_GROUP) {
    // Buffer to store the label
    char label[100+1];
    // Open the group
    hid_t group_id = H5Gopen2(gid, name, H5P_DEFAULT);
    if(group_id<0) cgp_doError;
    // Read the label
    err = get_str_attb(group_id, "label", 100, label);
    if(err!=0) cgp_doError;
    // Close the group
    herr = H5Gclose(group_id);
    if(herr<0) cgp_doError;
    // If the label is the same as the given label, increment the counter
    if(strcmp(label, data->label)==0) data->counter++;
}
return 0;

int num_nodes(hid_t pid, const char* label, int* num) {
    herr_t herr;
    // An iter_s to hold the iteration results
    struct iter_s data;
    // Initialize position to be -1
    data.pos = -1;
    // Initialize the counter to zero
    data.counter = 0;
    // Point the label to the given label
    data.label = (char*) label;

    // Iteration though the children of pid
    herr = H5Literate(pid, H5_INDEX_NAME, H5_ITER_INC, NULL, node_counter, &data);

    if(herr<0) cgp_doError;

    // Return the number of children whose label's match label
    *num = data.counter;
    return 0;

    herr_t node_idx_finder(hid_t gid, const char* name, const H5L_info_t* linfo, void *vdata) {
        int err = 0;
        herr_t herr;

        // Cast vdata as an iter_s
        struct iter_s* data = (struct iter_s*) vdata;

        // Info for each object
        H5O_info_t info;

        // Get the info for the current object
        herr = H5Oget_info_by_name(gid, name, &info, H5P_DEFAULT);
        if(herr<0) cgp_doError;

        // If the object is a group
        if(info.type==H5O_TYPE_GROUP) {
            // Buffer for label
            char label[100+1];

            // Buffer for name
            char nname[100+1];

            // Open the group
            hid_t group_id = H5Gopen2(gid, name, H5P_DEFAULT);
            if(group_id<0) cgp_doError;

            // Read the label
            err = get_str_attb(group_id, "label", 100, label);
            if(err!=0) cgp_doError;

            // Read the name
err = get_str_attb(group_id, "name", 100, nname);
if(err!=0) cgp_doError;
// Close the group
herr = H5Gclose(group_id);
if(herr<0) cgp_doError;
// If the labels are the same
if(strcmp(label,data->label)==0) {
    // If the names are the same, set the position to the counter
    if(strcmp(nname,data->name)==0) data->pos=data->counter;
    // Increment the counter
    data->counter++;
}
}
return 0;
}

int node_name2idx(hid_t pid, const char* label, const char* name, int* idx) {
herr_t herr;
// Data for iteration
struct iter_s data;
// Initialize position
data.pos = -1;
// Initialize counter
data.counter = 0;
// Point the label to the given label
data.label = (char*) label;
// Point the name to the given name
data.name = (char*) name;
// Iterate through all objects in this group
herr = H5Literate(pid, H5_INDEX_NAME, H5_ITER_INC, NULL, node_idx_finder, &data);
if(herr<0) cgp_doError;
// Set the index to pos
*idx = data.pos;
return 0;
herr_t node_name_finder(hid_t gid, const char* name, const H5L_info_t* linfo, void* vdata) {
    int err = 0;
    herr_t herr;
    // Cast vdata to an iter_s object
    struct iter_s* data = (struct iter_s*) vdata;
    // Info about the object
    H5O_info_t info;
    // Read the info
    herr = H5Oget_info_by_name(gid, name, &info, H5P_DEFAULT);
    if(herr<0) cgp_doError;
    // If the object is a group
    if(info.type==H5O_TYPE_GROUP) {
        // Buffer for the label
        char label[100+1];
        // Buffer for the name
        char nname[100+1];
        // Open the group
        hid_t group_id = H5Gopen2(gid, name, H5P_DEFAULT);
        if(group_id<0) cgp_doError;
        // Read the label
        err = get_str_attb(group_id, "label", 100, label);
        if(err!=0) cgp_doError;
        // Read the name
        err = get_str_attb(group_id, "name", 100, nname);
        if(err!=0) cgp_doError;
        // Close the group
        herr = H5Gclose(group_id);
        if(herr<0) cgp_doError;
        // If the label is the same as the given label
        if(strcmp(label, data->label)==0) {
            // If the counter is the same as the given coutner, copy the name
            if(data->counter==data->pos) strcpy(data->name, nname);
        }
// Increment the counter
data->counter++;
}

return 0;
}

int node_idx2name(hid_t pid, const char* label, int idx, char* name) {

herr_t herr;

// Data for iteration
struct iter_s data;

// Initialize the position
data.pos = idx;

// Initialize the counter
data.counter = 0;

// Point the label
data.label = (char*) label;

// Point the name
data.name = name;

// Iterate through the objects
herr = H5Literate(pid, H5_INDEX_NAME, H5_ITER_INC, NULL, node_name_finder, &data);

if(herr<0) cgp_doError;
return 0;
}

// hdf5_version_str
int hdf5_version_str(int len, char* buf) {

herr_t herr;

// Three components of the HDF5 version
unsigned int maj, min, rel;

// Read the version
herr = H5get_libversion(&maj, &min, &rel);

if(herr<0) cgp_doError;
// Preset the buffer to zeros
memset(buf,'\0',len+1);

// Write the version string to the buffer
sprintf(buf, "HDF5 Version %d.%d.%d", maj, min, rel);
return 0;

int hdf5_format_str(int len, char* buf) {
  herr_t herr;
  // Get a copy of the native float type
  hid_t type_id = H5Tcopy(H5T_NATIVE_FLOAT);
  if(type_id<0) cgp_doError;
  memset(buf,'\0',len+1);
  // Preset the buffer to zeros
  // Write the appropriate format string to the buffer
  if (H5Tequal(type_id, H5T_IEEE_F32BE)) strcpy(buf, "IEEE_BIG_32");
  else if (H5Tequal(type_id, H5T_IEEE_F32LE)) strcpy(buf, "IEEE_LITTLE_32");
  else if (H5Tequal(type_id, H5T_IEEE_F64BE)) strcpy(buf, "IEEE_BIG_64");
  else if (H5Tequal(type_id, H5T_IEEE_F64LE)) strcpy(buf, "IEEE_LITTLE_64");
  else sprintf(buf, "NATIVE_%d", H5Tget_precision(type_id));
  // Close the type
  herr = H5Tclose(type_id);
  if(herr<0) cgp_doError;
  return 0;
}

A.4.7 pcgns_util.h File Reference

#include "pcgnslib.h"
Include dependency graph for pcgns_util.h:

This graph shows which files directly or indirectly include this file:

Data Structures

- struct coords_s
  
  \textit{Struct to describe acoords node in the file.}

- struct section_s
  
  \textit{Struct to describe a section node in the file.}

- struct sol_s
  
  \textit{Struct to describe a solution node in the file.}

- struct zone_s
  
  \textit{Struct to describe a zone node in the file.}

- struct base_s
  
  \textit{Struct to describe a base node in the file.}
• struct file_s

    \textit{Struct to describe a CGNS file.}

• struct slice_s

\section*{Defines}

• \#define TRUE 1

• \#define FALSE 0

\section*{Typedefs}

• typedef struct coords_s coords_t

    \textit{Struct to describe a coords node in the file.}

• typedef struct section_s section_t

    \textit{Struct to describe a section node in the file.}

• typedef struct sol_s sol_t

    \textit{Struct to describe a solution node in the file.}

• typedef struct zone_s zone_t

    \textit{Struct to describe a zone node in the file.}

• typedef struct base_s base_t

    \textit{Struct to describe a base node in the file.}

• typedef struct file_s file_t
Struct to describe a CGNS file.

- typedef struct slice_s slice_t

**Functions**

- int next_file (int *fn)
- int free_file (file_t *file)
- int free_base (base_t *base)
- int free_zone (zone_t *zone)
- int free_coords (coords_t *coords)
- int free_section (section_t *section)
- int free_sol (sol_t *sol)
- int new_str (hid_t pid, const char *name, const char *value, int len)
- int get_str (hid_t pid, const char *name, int len, char *value)
- int new_str_attb (hid_t pid, const char *name, const char *value, int len)
- int get_str_attb (hid_t pid, const char *name, int len, char *value)
- int new_int (hid_t pid, const char *name, const int *value)
- int get_int (hid_t pid, const char *name, int *value)
- int new_int_attb (hid_t pid, const char *name, const int *value)
- int get_int_attb (hid_t pid, const char *name, int *value)
- int new_float (hid_t pid, const char *name, const float *value)
- int get_float (hid_t pid, const char *name, float *value)
- int new_float_attb (hid_t pid, const char *name, const float *value)
- int get_float_attb (hid_t pid, const char *name, float *value)
- int new_node (hid_t pid, const char *name, const char *label, const char *type)
- int del_node (hid_t pid, const char *name)
- int node_exists (hid_t pid, const char *name)
• int num_nodes (hid_t pid, const char *label, int *num)
• int node_name2idx (hid_t pid, const char *label, const char *name, int *idx)
• int node_idx2name (hid_t pid, const char *label, int idx, char *name)
• int hdf5_version_str (int len, char *buf)
• int hdf5_format_str (int len, char *buf)

Variables

• file_t * files

  Array of file_t’s which describe the open files.

• int files_count

  Internal count of used slots in {files}.

• int files_size

  Internal count of the slots in {files}.

• slice_t * write_queue

  Queue of IO write operations.

• int write_queue_len

  Length of write queue.

A.4.7.1 Detailed Description

Author:

Kyle Horne {horne.kyle@gmail.com}

Version:

0.2
A.4.7.2 LICENSE

BSD style license

A.4.7.3 DESCRIPTION

Header file for utility functions
Definition in file pcgns_util.h.

A.4.7.4 Define Documentation

A.4.7.4.1 #define FALSE 0

Definition at line 21 of file pcgns_util.h.

A.4.7.4.2 #define TRUE 1

Definition at line 17 of file pcgns_util.h.

A.4.7.5 Typedef Documentation

A.4.7.5.1 typedef struct base_s base_t

Struct to describe a base node in the file.

A.4.7.5.2 typedef struct coords_s coords_t

Struct to describe a coords node in the file.

A.4.7.5.3 typedef struct file_s file_t
Struct to describe a CGNS file.

A.4.7.5.4 typedef struct section_s section_t

Struct to describe a section node in the file.

A.4.7.5.5 typedef struct slice_s slice_t

A.4.7.5.6 typedef struct sol_s sol_t

Struct to describe a solution node in the file.

A.4.7.5.7 typedef struct zone_s zone_t

Struct to describe a zone node in the file.

A.4.7.6 Function Documentation

A.4.7.6.1 int del_node (hid_t pid, const char * name)

Delete a node with parent {pid}

Parameters:

pid [in]: HDF5 locator for parent

name [in]: Name of node

Returns:

Error code

Definition at line 506 of file pcgns_util.c.
Here is the caller graph for this function:

```
free_base  cgp_base_write  main
    cgp_close
    cgp_zone_write
    cgp_section_write
    cgp_coord_write
    cgp_sol_write
    cgp_base_write
    cgp_close
```

A.4.7.6.2 int free_base (base_t * base)

Free the memory of a base_t object

Parameters:

*base [in]: Pointer the base

Returns:

Error code

Definition at line 103 of file pcgns_util.c.

Here is the call graph for this function:

```
free_base  free_coord
    free_zone  free_section
    free_sol
```

Here is the caller graph for this function:

```
free_base  cgp_base_write  main
    free_file  cgp_close
```

Here is the caller graph for this function:

```
free_base  cgp_base_write  main
    free_file  cgp_close
```

A.4.7.6.2 int free_base (base_t * base)

Free the memory of a base_t object

Parameters:

*base [in]: Pointer the base

Returns:

Error code

Definition at line 103 of file pcgns_util.c.

Here is the call graph for this function:

```
free_base  free_coord
    free_zone  free_section
    free_sol
```

Here is the caller graph for this function:

```
free_base  cgp_base_write  main
    free_file  cgp_close
```
A.4.7.6.3  int free_coords (coords_t * coords)

Free the memory of a coords_t object

Parameters:

coords  [in]: Pointer the coords

Returns:

Error code

A.4.7.6.4  int free_file (file_t * file)

Free the memory of a file_t object

Parameters:

file  [in]: Pointer the file

Returns:

Error code

Definition at line 69 of file pcgns_util.c.

Here is the call graph for this function:

Here is the caller graph for this function:

A.4.7.6.5  int free_section (section_t * section)

Free the memory of a section_t object
Parameters:

section [in]: Pointer the section

Returns:

Error code

Definition at line 163 of file pcgns_util.c.

Here is the caller graph for this function:

```
free_section  free_zone  free_base  cgp_zone_write  main
          cgp_base_write  free_file  cgp_close
```

A.4.7.6.6 int free_sol (sol_t * sol)

Free the memory of a sol_t object

Parameters:

sol [in]: Pointer the sol

Returns:

Error code

Definition at line 179 of file pcgns_util.c.

Here is the caller graph for this function:

```
free_sol  free_zone  free_base  cgp_zone_write  main
          cgp_base_write  free_file  cgp_close
```

A.4.7.6.7 int free_zone (zone_t * zone)

Free the memory of a zone_t object
Parameters:

zone [in]: Pointer the zone

Returns:

Error code

Definition at line 121 of file pcgns_util.c.

Here is the call graph for this function:

Here is the caller graph for this function:

A.4.7.6.8 int get_float (hid_t pid, const char * name, float * value)

Read an float with parent {pid}, name {name}, and value {value}

Parameters:

pid [in]: HDF5 locator for parent

name [in]: Name of float

value [out]: Float to read

Returns:

Error code

Definition at line 412 of file pcgns_util.c.
A.4.7.6.9  int get_float_attb (hid_t pid, const char * name, float * value)

Read an float attribute at location {pid} with {name} with {value}

Parameters:

  pid [in]: HDF5 locator for parent

  name [in]: Name of float

  value [out]: Float to read

Returns:

Error code

Definition at line 456 of file pcgns_util.c.

A.4.7.6.10  int get_int (hid_t pid, const char * name, int * value)

Read an integer with parent {pid}, name {name}, and value {value}

Parameters:

  pid [in]: HDF5 locator for parent

  name [in]: Name of integer

  value [out]: Integer to read

Returns:

Error code

Definition at line 322 of file pcgns_util.c.

A.4.7.6.11  int get_int_attb (hid_t pid, const char * name, int * value)

Read an integer attribute at location {pid} with {name} with {value}

Parameters:

  pid [in]: HDF5 locator for parent
**name** [in]: Name of integer

**value** [out]: Integer to read

**Returns:**

Error code

Definition at line 366 of file pcgns_util.c.

**A.4.7.6.12 int get_str (hid_t pid, const char *name, int len, char *value)**

Read a string with parent {pid}, name {name}, and value {value}

**Parameters:**

- **pid** [in]: HDF5 locator for parent
- **name** [in]: Name of string
- **len** [in]: Length of string in file
- **value** [out]: Contents of string

**Returns:**

Error code

Definition at line 214 of file pcgns_util.c.

Here is the caller graph for this function:

![Caller Graph](image)

**A.4.7.6.13 int get_str_at ttb (hid_t pid, const char *name, int len, char *value)**

Read a string attribute at location {pid} with {name} with {value}

**Parameters:**

- **pid** [in]: HDF5 locator for parent
- **name** [in]: Name of string
**len** [in]: Length of string in file

**value** [out]: Contents of string

**Returns:**

Error code

Definition at line 269 of file pcgns_util.c.

Here is the caller graph for this function:

---

**A.4.7.6.14 int hdf5_format_str (int len, char * buf)**

Fill a buffer with the HDF5 format string

**Parameters:**

**len** [in]: Length of buf

**buf** [out]: String with the HDF5 format

**Returns:**

Error code

Definition at line 706 of file pcgns_util.c.

Here is the caller graph for this function:

---

**A.4.7.6.15 int hdf5_version_str (int len, char * buf)**

Fill a buffer with the HDF5 version number
Parameters:

\( len \) [in]: Length of buf

\( buf \) [out]: String with HDF5 version number

Returns:

Error code

Definition at line 692 of file `pcgns_util.c`.

Here is the caller graph for this function:

```
| hdf5_version_str | cgp_open | main |
```

A.4.7.6.16 int new_float (hid_t \( pid \), const char * \( name \), const float * \( value \))

Create a new float with parent \( \{pid\} \), name \( \{name\} \), and value \( \{value\} \)

Parameters:

\( pid \) [in]: HDF5 locator for parent

\( name \) [in]: Name of float

\( value \) [in]: Float to write

Returns:

Error code

Definition at line 390 of file `pcgns_util.c`.

Here is the caller graph for this function:

```
| new_float | cgp_open | main |
```

A.4.7.6.17 int new_float_attb (hid_t \( pid \), const char * \( name \), const float * \( value \))

Create a new float attribute at location \( \{pid\} \) with \( \{name\} \) with \( \{value\} \)
Parameters:

- **pid** [in]: HDF5 locator for parent
- **name** [in]: Name of float
- **value** [in]: Float to write

Returns:

Error code

Definition at line 434 of file pcgns_util.c.

A.4.7.6.18 \texttt{int new\_int (hid\_t pid, const char \* name, const int \* value)}

Create a new integer with parent \{pid\}, name \{name\}, and value \{value\}

Parameters:

- **pid** [in]: HDF5 locator for parent
- **name** [in]: Name of integer
- **value** [in]: Integer to write

Returns:

Error code

Definition at line 300 of file pcgns_util.c.

Here is the caller graph for this function:

![Caller Graph](image)

A.4.7.6.19 \texttt{int new\_int\_attb (hid\_t pid, const char \* name, const int \* value)}

Create a new integer attribute at location \{pid\} with \{name\} with \{value\}

Parameters:

- **pid** [in]: HDF5 locator for parent
**name** [in]: Name of integer

**value** [in]: Integer to write

**Returns:**

Error code

Definition at line 344 of file pegns_util.c.

Here is the caller graph for this function:

```
A.4.7.6.20 int new_node (hid_t pid, const char *name, const char *label, const char *type)
```

Create a new node at with parent {pid}

**Parameters:**

- **pid** [in]: HDF5 locator for parent
- **name** [in]: Name of node
- **label** [in]: Label of node
- **type** [in]: Type of node

**Returns:**

Error code
A.4.7.6.21  int new_str (hid_t pid, const char * name, const char * value, int len)

Create a new string with parent {pid}, name {name}, and value {value}

Parameters:

  pid [in]: HDF5 locator for parent

  name [in]: Name of string

  value [in]: Contents of string

  len [in]: Length of string in file

Returns:

  Error code

Definition at line 192 of file p cgns_util.c.
Here is the caller graph for this function:

A.4.7.6.22  \textbf{int new\_str\_attb (hid\_t \textit{pid}, const char \* \textit{name}, const char \* \textit{value}, int \textit{len})}

Create a new string attribute at location \{pid\} with \{name\} with \{value\}

**Parameters:**

- \textit{pid}  [in]: HDF5 locator for parent
- \textit{name}  [in]: Name of string
- \textit{value}  [in]: Contents of string
- \textit{len}  [in]: Length of string in file

**Returns:**

Error code

Definition at line 239 of file pcgns_util.c.

Here is the caller graph for this function:
**A.4.7.6.23 int next_file (int *fn)**

Return the number of the next available file_t in {files}

**Parameters:**

*fn* [out]: Index of next file

**Returns:**

Error code

Definition at line 37 of file pcgns_util.c.

Here is the call graph for this function:

![Call Graph](next_file -> cleanup_files)

Here is the caller graph for this function:

![Caller Graph](next_file -> cg_open -> main)

**A.4.7.6.24 int node_exists (hid_t pid, const char *name)**

Check if a node exists with name {name} and parent {pid}

**Parameters:**

*pid* [in]: HDF5 locator for parent

*name* [in]: Name of node

**Returns:**

{1=Exists, 0=Does not exist}

Definition at line 514 of file pcgns_util.c.
A.4.7.6.25  int node_idx2name (hid_t \textit{pid}, const char *\textit{label}, int \textit{idx}, char *\textit{name})

Convert a node id to a node name

**Parameters:**

\begin{itemize}
  \item \textit{pid} [in]: HDF5 locator for parent
  \item \textit{label} [in]: Label of node
  \item \textit{idx} [in]: Index of node
  \item \textit{name} [out]: Name of node
\end{itemize}

Definition at line 672 of file pcgns_util.c.

Here is the call graph for this function:

```
A.4.7.6.25  int node_idx2name (hid_t \textit{pid}, const char *\textit{label}, int \textit{idx}, char *\textit{name})
```

Here is the caller graph for this function:
A.4.7.6.26 int node_name2idx (hid_t pid, const char *label, const char *name, int *idx)

Convert a node name to a node id

Parameters:

  pid [in]: HDF5 locator for parent

  label [in]: Label of node

  name [in]: Name of node

  idx [out]: Index of node

Returns:

Error code

Definition at line 612 of file pcgns_util.c.
Here is the call graph for this function:

A.4.7.6.27 int num_nodes (hid_t pid, const char *label, int *num)

Count the number of nodes of type {label} with parent {pid}

Parameters:

  pid [in]: HDF5 locator for parent

  label [in]: Label of nodes

  num [out]: Number of nodes

Returns:

Error code

Definition at line 554 of file pcgns_util.c.
Here is the call graph for this function:
Here is the caller graph for this function:

A.4.7.7 Variable Documentation

A.4.7.7.1 file_t* files

Array of file_t’s which describe the open files.
Definition at line 21 of file pcgns_util.c.

A.4.7.7.2 int files_count

Internal count of used slots in {files}.
Definition at line 22 of file pcgns_util.c.

A.4.7.7.3 int files_size

Internal count of the slots in {files}.
Definition at line 23 of file pcgns_util.c.

A.4.7.7.4 slice_t* write_queue

Queue of IO write operations.
Definition at line 25 of file pcgns_util.c.
A.4.7.7.5 int write_queue_len

Length of write queue.
Definition at line 26 of file pcgns_util.c.

A.4.8 pcgns_util.h

0001
0002
0003
0004
0005
0006
0007
0008
0009
0010
0011  ifndef PCGNS_UTIL_H_
0012  #define PCGNS_UTIL_H_
0013
0014  #include "pcgnslib.h"
0015
0016  ifndef TRUE
0017  #define TRUE 1
0018  #endif
0019
0020  ifndef FALSE
0021  #define FALSE 0
0022  #endif
0023
0024
0025  //---------------------
0026  /// Begin Datatypes ///
0027  //---------------------
typedef struct coords_s {
  int idx;
  char coordname[100+1];
  hid_t group_id;
} coords_t;

typedef struct section_s {
  int idx;
  char sectionname[100+1];
  hid_t group_id;
  ElementType_t type;
  int range[2];
  hid_t connectivity_id;
  hid_t range_id;
} section_t;

typedef struct sol_s {
  int idx;
  char solname[100+1];
  hid_t group_id;
} sol_t;

typedef struct zone_s {
  int idx;
  char zonename[100+1];
  coords_t* coords;
  int ncoords;
  section_t* sections;
  int nsections;
  sol_t* sols;
  int nsols;
  hid_t group_id;
  int* nijk;
  ZoneType_t type;
typedef struct base_s {
    int idx;
    char basename[100+1];
    zone_t *zones;
    int nzones;
    hid_t group_id;
    int cell_dim;
    int phys_dim;
} base_t;

typedef struct file_s {
    int idx;
    char filename[100+1];
    int isOpen;
    base_t *bases;
    int nbases;
    hid_t file_id;
    hid_t plist_id;
    MPI_Comm comm;
    MPI_Info info;
    int rank;
    int size;
} file_t;

typedef struct slice_s {
    SliceType_t type;
    int rank;
    int* min;
    int* max;
    void* data;
    int F;

int B;
int Z;
int Selector;
char name[100+1];
} slice_t;

//==================================//
//== Begin Global Data Prototypes ==//
//=================================//
extern file_t* files;
extern int files_count;
extern int files_size;
extern slice_t* write_queue;
extern int write_queue_len;

//===============================//
//== Begin Function Prototypes ==//
//===============================//
int next_file(int* fn);
int free_file(file_t* file);
int free_base(base_t* base);
int free_zone(zone_t* zone);
int free_coords(coords_t* coords);
int free_section(section_t* section);
int free_sol(sol_t* sol);

int new_str(hid_t pid, const char* name, const char* value, int len);

int new_str_attb(hid_t pid, const char* name, const char* value, int len);

int new_int(hid_t pid, const char* name, const int* value);

int new_int_attb(hid_t pid, const char* name, const int* value);

int new_float(hid_t pid, const char* name, const float* value);

int new_float_attb(hid_t pid, const char* name, const float* value);

int new_double(hid_t pid, const char* name, const double* value);

int new_double_attb(hid_t pid, const char* name, const double* value);
A.4.9  pcgnslib.c File Reference

#include "pcgnslib.h"
#include "pcgns_util.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
Include dependency graph for pcgnslib.c:

Functions

- int cgp_open (const char *filename, int mode, MPI_Comm comm, MPI_Info *info, int *fn)
- int cgp_close (int fn)
- int cgp_base_read (int fn, int B, char *basename, int *cell_dim, int *phys_dim)
- int cgp_base_write (int fn, char const *basename, int cell_dim, int phys_dim, int *B)
- int cgp_nbases (int fn, int *nbases)
- int cgp_zone_read (int fn, int B, int Z, char *zonename, int *nijk)
- int cgp_zone_type (int fn, int B, int Z, ZoneType_t *zonetype)
- int cgp_zone_write (int fn, int B, const char *zonename, const int *nijk, ZoneType_t type, int *Z)
- int cgp_nzones (int fn, int B, int *nzones)
- int cgp_coord_write (int fn, int B, int Z, DataType_t type, const char *coordname, int *C)
- int cgp_coord_write_data (int fn, int B, int Z, int C, int *min, int *max, void *coord_array)
- int cgp_sol_write_data (int fn, int B, int Z, int S, int *min, int *max, void *data)
- int cgp_sol_write (int fn, int B, int Z, char *solname, GridLocation_t location, int *S)
- int cgp_nsols (int fn, int B, int Z, int *nsols)
- int cgp_section_write (int fn, int B, int Z, char *sectionname, ElementType_t type, int start, int end, int nbndry, int *S)
- int cgp_section_write_data (int fn, int B, int Z, int S, int min, int max, int *elements)
• int cgp_array_write (int fn, int B, int Z, char *arrayname, GridLocation_t location)
• int cgp_array_write_data (int fn, int B, int Z, char *arrayname, int *min, int *max, void *data)
• int queue_slice_write (SliceType_t type, int F, int B, int Z, void *SN, int rank, int *min, int *max, void *data)
• int queue_flush (void)

Variables

• int node_counts [24]
• int preallocate = 0

A.4.9.1 Detailed Description

Author:

Kyle Horne <horne.kyle@gmail.com>

Version:

0.2

A.4.9.2 LICENSE

BSD style license

A.4.9.3 DESCRIPTION

Implimentation of functions provided by the pcgns library

Definition in file pcgnslib.c.

A.4.9.4 Function Documentation

A.4.9.4.1 int cgp_array_write (int fn, int B, int Z, char *arrayname, GridLocation_t

location)

Write an array to a zone
Parameters:

\( fn \) [in]: Handle of the file

\( B \) [in]: Index of the base

\( Z \) [in]: Index of the zone

\( \text{arrayname} \) [in]: Name of array

\( \text{location} \) [in]: Location of solution within each cell

Returns:

Error code

Definition at line 1191 of file pcgslib.c.

Here is the call graph for this function:

Here is the caller graph for this function:

A.4.9.4.2 \textbf{int cgp\_array\_write\_data (int } fn, \textbf{int } B, \textbf{int } Z, \textbf{char } * \text{arrayname}, \textbf{int } * \text{min}, \textbf{int } * \text{max}, \textbf{void } * \text{data)}

Write an array’s data in parallel

Parameters:

\( fn \) [in]: Handle of the file

\( B \) [in]: Index of the base

\( Z \) [in]: Index of the zone
arrayname [in]: Name of array

min [in]: Lower bound array for data

max [in]: Upper bound array for data

data [in]: Data to be written

Returns:

Error code

Definition at line 1251 of file pcgnslib.c.

Here is the caller graph for this function:

```
cgp_array_write_data
    queue_flush
    main
```

A.4.9.4.3 int cgp_base_read (int fn, int B, char * basename, int * cell_dim, int * phys_dim)

Read info about a base

Parameters:

fn [in]: Handle of the file

B [in]: Index of the base

basename [out]: Name of the base

cell_dim [out]: Cell dimensions of the base

phys_dim [out]: Physical dimensions of the base

Returns:

Error code

Definition at line 169 of file pcgnslib.c.

Here is the call graph for this function:

```
cgp_base_read
    node_idx2name
    num_nodes
    node_name_finder
    get_str_attb
    node_counter
```

```
Here is the caller graph for this function:

A.4.9.4.4 int cgp_base_write (int fn, char const * basename, int cell_dim, int phys_dim, int * B)

Write a base to a file

Parameters:

fn  int[in]: Handle f the file

basename  [in]: Name of the base to write

cell_dim  [in]: Cell dimensions of the base

phys_dim  [in]: Physical dimensions of the base

B  [out]: Index of the base

Returns:

Error code

Definition at line 234 of file pcgnslib.c.

Here is the call graph for this function:

Here is the caller graph for this function:
A.4.9.4.5 int cgp_close (int fn)

Close a previously opened file

Parameters:

fn [in]: Handle of the file to close

Returns:

Error code

Definition at line 159 of file pcgnslib.c.

Here is the call graph for this function:

Here is the caller graph for this function:

A.4.9.4.6 int cgp_coord_write (int fn, int B, int Z, DataType_t type, const char *coordname, int *C)

Write coords group, but not data, to a grid

Parameters:

fn [in]: Handle of the file

B [in]: Index of the base

Z [in]: Index of the zone

data_type [in]: Type of floats stored

coordname [in]: Name of the coords
C [out]: Index of the coords

Definition at line 667 of file pcgnslib.c.

Here is the call graph for this function:

Here is the caller graph for this function:

A.4.9.4.7 int cgp_coord_write_data (int fn, int B, int Z, int C, int * min, int * max, void * coord_array)

Write coords to a grid in parallel

Parameters:

fn [in]: Handle of the file

B [in]: Index of the base

Z [in]: Index of the zone

C [in]: Index of the coords

range_min [in]: Array of lower bound index

range_max [in]: Array of upper bound index

coord_array [in]: Pointer to the data

Returns:

Error code

Definition at line 750 of file pcgnslib.c.
Here is the caller graph for this function:

![Caller Graph]

### A.4.9.4.8 int cgp_nbases (int fn, int *nbases)

Read the number of bases in a file

**Parameters:**

- **fn** [in]: Handle of the file
- **nbases** [out]: Number of bases in the specified file

**Returns:**

Error code

Definition at line 318 of file pcgnslib.c.

### A.4.9.4.9 int cgp_nsols (int fn, int B, int Z, int *nsols)

Read the number of solutions in a zone

**Parameters:**

- **fn** [in]: Handle of the file
- **B** [in]: Index of the base
- **Z** [in]: Index of the zone
- **nsols** [out]: Number of solutions in the specified zone

**Returns:**

Error code

Definition at line 960 of file pcgnslib.c.
A.4.9.4.10  int cgp_nzones (int fn, int B, int * nzones)

Read the number of zones in a base

Parameters:

  fn  [in]: Handle of file

  B   [in]: Index of base

  nzones  [out]: Number of zones in the specified base

Returns:

  Error code

Definition at line 660 of file pcgnslib.c.

A.4.9.4.11  int cgp_open (const char * filename, int mode, MPI_Comm comm, MPI_Info * info, int * fn)

Open a file for reading and writing

Parameters:

  filename  [in]: Name of the file to open

  mode  [in]: IO mode (read/write)

  comm  [in]: MPI communicator on which to open the file

  info  [in]: MPI info object to allow hints passed to MPI-IO

  fn  [out]: Handle of the opened file

Returns:

  Error code

Definition at line 42 of file pcgnslib.c.
A.4.9.4.12 int cgp_section_write (int fn, int B, int Z, char * sectionname, ElementType_t type, int start, int end, int nbndry, int * S)

Write the element connectivity groups for a section

Parameters:

fn [in]: Handle of the file

B [in]: Index of the base

Z [in]: Index of the zone

C [in]: Index of the coords

sectionname [in]: Name of element section

type [in]: Type of element data

start [in]: Element lower bound index
end [in]: Element upper bound index

nbndry [in]: Number of boundary elements (unused)

S [out]: Section index

**Returns:**

Error code

Definition at line 976 of file pegnslib.c.

Here is the call graph for this function:

![Call Graph](image)

Here is the caller graph for this function:

![Caller Graph](image)

A.4.9.4.13 int cgp_section_write_data (int fn, int B, int Z, int S, int min, int max, int *elements)

Write the element connectivity data for a section

**Parameters:**

fn [in]: Handle of the file

B [in]: Index of the base

Z [in]: Index of the zone

C [in]: Index of the coords

S [in]: Section index

min [in]: Output array lower bound index
**max** [in]: Output array upper bound index

**elements** [in]: Pointer to the data

**Returns:**

Error code

Definition at line 1126 of file pcgnslib.c.

Here is the caller graph for this function:

```
cgp_section_write_data
```

A.4.9.4.14  int cgpsol_write (int fn, int B, int Z, char *solname, GridLocation_t location,
int *S)

Write a solution to a zone

**Parameters:**

*fn* [in]: Handle of the file

*B* [in]: Index of the base

*Z* [in]: Index of the zone

*solname* [in]: Name of solution

*location* [in]: Location of solution within each cell

*S* [out]: Index of solution

**Returns:**

Error code

Definition at line 876 of file pcgnslib.c.
Here is the call graph for this function:

Here is the caller graph for this function:

A.4.9.4.15  int cgpsol_write_data (int fn, int B, int Z, int S, int * min, int * max, void * data)  

Write a solution’s data in parallel  

Parameters:

fn  [in]: Handle of the file  
B  [in]: Index of the base  
Z  [in]: Index of the zone  
S  [in]: Index of solution  
min  [in]: Lower bound array for data  
max  [in]: Upper bound array for data  

data  [in]: Data to be written  

Returns:

Error code

Definition at line 813 of file pcgnslib.c.

Here is the caller graph for this function:
A.4.9.4.16  int cgp_zone_read (int fn, int B, int Z, char *zonename, int *nijk)

Parameters:

fn [in]: Handle of the file
B [in]: Index of the base
Z [in]: Index of the zone to read
zonename [out]: Name of the zone
nijk [out]: Dimensions of the zone

Returns:

Error code

Definition at line 325 of file pcnslib.c.

Here is the call graph for this function:

Here is the caller graph for this function:

A.4.9.4.17  int cgp_zone_type (int fn, int B, int Z, ZoneType_t *zonetype)

Parameters:

fn [in]: Handle of the file
$B$ [in]: Index of the base

$Z$ [in]: Index of the zone

$zonetype$ [out]: Type of zone

**Returns:**

Error code

Definition at line 535 of file pcgnslib.c.

**A.4.9.4.18** \[\text{int cgp\_zone\_write (int } fn, \text{ int } B, \text{ const char } * zonename, \text{ const int } * nijk, \text{ ZoneType\_t type, int } * Z)\]

Write a zone to a base

**Parameters:**

$fn$ [in]: Handle of the file

$B$ [in]: Index of the base

$zonename$ [in]: Name of the zone to write

$nijk$ [in]: Dimensions of the zone

$type$ [in]: Type of zone

$Z$ [out]: Index of the zone

**Returns:**

Error code

Definition at line 544 of file pcgnslib.c.
Here is the call graph for this function:

Here is the caller graph for this function:

A.4.9.4.19  int queue_flush (void)

Flush all the IO operations waiting in the queue

Returns:

Error code

Definition at line 1344 of file pcgnslib.c.

Here is the call graph for this function:

Here is the caller graph for this function:
int queue_slice_write (SliceType_t type, int F, int B, int Z, void *SN, int rank, int *min, int *max, void *data)

Queue an IO write operation for flushing later

Parameters:

- **type** [in]: Type of operation to queue
- **F** [in]: Handle of the file
- **B** [in]: Index of the base
- **Z** [in]: Index of the zone
- **SN** [in]: Pointer to array locator, which is an int for coordinates, solutions and sections, but a string for arrays
- **rank** [in]: Rank of data to be written
- **min** [in]: Pointer to the minimum location array
- **max** [in]: Pointer to the maximum location array
- **data** [in]: Pointer to the data to be written

Returns:

Error code

Definition at line 1315 of file pcgnslib.c.

Here is the caller graph for this function:

![Caller Graph](image)

A.4.9.5 Variable Documentation

A.4.9.5.1 int node_counts[24]

Initial value:
{ -1, -1, 1, 2, 3, 3, 6, 4, 8, 9, 4, 10, 5, 13, 14, 6, 15, 18, 8, 20, 27, -1, -1, -1, -1 }

Definition at line 21 of file pcgnslib.c.

A.4.9.5.2 int preallocate = 0

Definition at line 35 of file pcgnslib.c.

A.4.10 pcgnslib.c

00001
00002
00003
00004
00005
00006
00007
00008
00009
00010
00011 #include "pcgnslib.h"
00012 #include "pcgns_util.h"
00013
00014 #include "stdio.h"
00015 #include "stdlib.h"
00016 #include "string.h"
00017
00018 // Number of nodes per element
00019 // Index corresponds to the ElementType_t enum
00020 // Negative numbers indicate non-supported types
00021 int node_counts[24] = {
00022 -1, -1,
00023 1,
00024 2,3,
00025 3,6,
00026 4,8,9,
00027 4,10,
00028 5,13,14,
00029 6,15,18,
00030 8,20,27,
00031 -1,
00032 -1,-1
00033 };
00034
00035 int preallocate = 0;
00036
00037 //================================//
00038 //== Begin Function Definitions ==//
00039 //================================//
00040
00041 //== File IO Prototypes ==//
00042 int cgp_open(const char* filename, int mode, MPI_Comm comm, MPI_Info* info, int* fn) {
00043     int err;
00044     herr_t herr;
00045     // Test file to check for file's existance
00046     FILE* test_file = fopen(filename, "rb");
00047     // Flag for file's existance
int file_exists = (test_file==NULL)?FALSE:TRUE;

// Flag for file’s HDF5 status
int file_isHDF5 = FALSE;

// If the file exists
if(file_exists) {
    // Close the file
close(test_file);
    // Test to see if it is an HDF5 file
    file_isHDF5 = H5Fis_hdf5(filename)?TRUE:FALSE;
}

// Create a new file reference in the global files array
next_file(fn);

// Create a pointer to this file in that array
file_t* file = &(files[*fn]);

// Set the file’s index
file->idx = *fn;

// Set the file’s state to open
file->isOpen = TRUE;

// Set the file’s name
strcpy(file->filename,filename);

// Set the file’s communicator
file->comm = comm;

// Set the MPI_Info for the file
err = MPI_Info_dup(*info,&(file->info));

if(err!=MPI_SUCCESS) cgp_doError;

// Set the rank on this communicator
err = MPI_Comm_rank(comm,&(file->rank));

if(err!=MPI_SUCCESS) cgp_doError;

// Set the size of this communicator
err = MPI_Comm_size(comm,&(file->size));

if(err!=MPI_SUCCESS) cgp_doError;

// Set the access property list
file->plist_id = H5Pcreate(H5P_FILE_ACCESS);

if(file->plist_id<0) cgp_doError;

// Set the access property list to use MPI
herr = H5Pset_fapl_mpio(file->plist_id, file->comm, file->info);
if(herr<0) cgp_doError;
// If the file exists and it is an HDF5 file, try to read is as CGNS
if(file_exists&&file_isHDF5) {
    // Open the file with HDF5 and set the file_id
    file->file_id = H5Fopen(filename, H5F_ACC_RDWR, file->plist_id);
    if(file->file_id<0) cgp_doError;
    // Read the number of bases in the file
    num_nodes(file->file_id, "CGNSBase_t", &(file->nbases));
    // Allocate space to store the bases
    file->bases = malloc(file->nbases*sizeof(base_t));
    if(file->bases==NULL) cgp_doError;
    // Loop counter
    int k;
    // For each base in bases...
    for(k=0;k<file->nbases;k++) {
        // Buffer to store the name of the base
        char basename[100+1];
        // Find the name of the k’th base in bases
        node_idx2name(file->file_id, "CGNSBase_t", k, basename);
        // Open that base
        file->bases[k].group_id = H5Gopen2(file->file_id, basename, H5P_DEFAULT);
        if(file->bases[k].group_id<0) cgp_doError;
        // Set the base’s index
        file->bases[k].idx = k;
        // Copy the base’s name
        strcpy(file->bases[k].basename, basename);
        // Initialize it to have no zones
        file->bases[k].zones = NULL;
        file->bases[k].nzones = 0;
    }
}
// If the file does not exist, create an empty CGNS file
else if((!file_exists)||(file_isHDF5)) {
// Create a new HDF5 file
file->file_id = H5Fcreate(filename, H5F_ACC_TRUNC, H5P_DEFAULT, file->
plist_id);

if(file->file_id<0) cgp_doError;

// Create the needed attributes to describe the root node
new_str_attb(file->file_id, "label", "Root Node of ADF File", 32);
new_str_attb(file->file_id, "name", "HDF5 MotherNode", 32);
new_str_attb(file->file_id, "type", "MT", 2);

// Buffer for version information
char version[100+1];

// Buffer for format information
char format[100+1];

// Get the HDF5 format string
hdf5_format_str(100, format);

// Get the HDF5 version string
hdf5_version_str(100, version);

// Write the HDF5 format string to the file
new_str(file->file_id, " format", format, strlen(format));

// Write the HDF5 version string to the file
new_str(file->file_id, " hdf5version", version, 32);

// Create a CGNS library version number
new_node(file->file_id, "CGNSLibraryVersion", "CGNSLibraryVersion_t", "R4")
;

// Open that version number group
hid_t group_id = H5Gopen2(file->file_id, "CGNSLibraryVersion", H5P_DEFAULT)
;

if(group_id<0) cgp_doError;

// Set the files to be compatible with CGNS version 3.0
float libversion = 3.0;

// Write the version number to the file
new_float(group_id, " data", &libversion);

// Close the version number group
herr = H5Gclose(group_id);

if(herr<0) cgp_doError;
// set the file to have no bases
file->bases = NULL;
file->nbases = 0;
int dummy = 0;
err = new_int(file->file_id, "dummy", &dummy);
if(err!=0) cgp_doError;
return 0;
}

int cgp_close(int fn) {
    // Free the file referenced by fn
    printTime;
del_node(files[fn].file_id, "dummy");
    free_file(&files[fn]);
    printTime;
    return 0;
}

//= Base IO Prototypes =/
int cgp_base_read(int fn, int B, char* basename, int* cell_dim, int* phys_dim) {
    if(fn>=files_count||fn<0) cgp_doError;
    if(!files[fn].isOpen) cgp_doError;
    if(B>files[fn].nbases||B<=0) cgp_doError;
    herr_t herr;
    // Pointer to the base being read
    base_t* base = &(files[fn].bases[B-1]);
    // Get the name of the base
    strcpy(basename, base->basename);
    // Size of array to read
    hsize_t dim = 2;
    // Create shape object for shape of array
    hid_t shape_id = H5Screate_simple(1,&dim,NULL);
    if(shape_id<0) cgp_doError;
// Open the data array in the file
hid_t data_id = H5Dopen2(base->group_id, "data", H5P_DEFAULT);
if(data_id<0) cgp_doError;
// Buffer to store the data
int data[2];
// Read the data array containing this base’s dimensions
herr = H5Dread(data_id, H5T_NATIVE_INT, shape_id, shape_id, H5P_DEFAULT, data);
if(herr<0) cgp_doError;
// Copy the data out of the array [The order here needs verification]
*cell_dim = data[0];
base->cell_dim = data[0];
*phys_dim = data[1];
base->phys_dim = data[1];
// Close the data array in the file
herr = H5Dclose(data_id);
if(herr<0) cgp_doError;
// Destroy the shape object
herr = H5Sclose(shape_id);
if(herr<0) cgp_doError;

// Read Zone info
// Read the number of zones in the base
num_nodes(base->group_id, "Zone_t", &base->nzones);
// Allocate space for zone descriptors
base->zones = (zone_t *) malloc(base->nzones*sizeof(zone_t));
if(base->zones==NULL) cgp_doError;
// Loop index
int k;
// For each zone in zones
for(k=0;k<base->nzones;k++) {
  // Buffer to store the zone’s name
  char zonename[100+1];
  // Get the name of the k’th zone
  node_idx2name(base->group_id, "Zone_t", k, zonename);
// Open the group for this zone
    base->zones[k].group_id = H5Gopen2(base->group_id, zonename, H5P_DEFAULT);
    if (base->zones[k].group_id < 0) cgp_doError;

    // Set the zone’s index
    base->zones[k].idx = k;

    // Copy the zone’s name
    strcpy(base->zones[k].zonename, zonename);

    // Default to no coords or solutions
    base->zones[k].coords = NULL;
    base->zones[k].ncoords = 0;
    base->zones[k].sols = NULL;
    base->zones[k].nsols = 0;
}

return 0;

}

int cgp_base_write(int fn, char const* basename, int cell_dim, int phys_dim, int* B) {
    if (fn >= files_count || fn < 0) cgp_doError;
    if (!files[fn].isOpen) cgp_doError;

    herr_t herr;

    // Pointer to this file
    file_t* file = &(files[fn]);

    // Index
    int idx;

    // If this base already exists, replace the old one
    if (node_exists(file->file_id, basename)) {
        // Loop index
        int k;

        // Find this base in the list of bases for this file
        for (k = 0; k < file->nbases; k++) if (strcmp(basename, file->bases[k].basename) == 0)
            idx = k;

        // Free the old base in memory
        free_base(&(file->bases[idx]));

        // Delete the old base in the file
    }
```c
    del_node(file->file_id, basename);

    // If this base does not yet exist, create one
    else {
        // Loop index
        int k;
        // Allocate space for a longer list of bases
        base_t* bases = malloc((file->nbases+1) * sizeof(base_t));
        if(bases==NULL) cgp_doError;
        // Copy all the old bases to the new list
        for(k=0;k<file->nbases;k++) bases[k] = file->bases[k];
        // Free the old list
        if(file->bases!=NULL) free(file->bases);
        // Point the file to use the new list
        file->bases = bases;
        // Increment the number of bases in the file
        file->nbases++;
        // Set the index to the last base in the list
        idx = file->nbases-1;
    }
    // Pointer to the current base
    base_t* base = &file->bases[idx]);
    // Copy the name of the base
    strcpy(file->bases[idx].basename, basename);
    // Set the index of the base
    base->idx = idx;
    // Create no zones in the base
    base->zones = NULL;
    base->nzones = 0;
    // Set the base’s dimensions
    base->cell_dim = cell_dim;
    base->phys_dim = phys_dim;
    // Return the index of the base
    *B = idx+1;
    // Create the base in the file
```
new_node(file->file_id, basename, "CGNSBase_t", "I4");

// Open the base
hid_t group_id = H5Gopen2(file->file_id, basename, H5P_DEFAULT);
if (group_id<0) cgp_doError;

// Copy the group_id for later use
file->bases[idx].group_id = group_id;

// Set the size of the array to write
hsize_t dim = 2;

// Create a shape object for the array to write
hid_t shape_id = H5Screate_simple(1,&dim,NULL);
if (shape_id<0) cgp_doError;

// Create the data in the file
hid_t data_id = H5Dcreate2(group_id, " data", H5T_NATIVE_INT, shape_id, H5P_DEFAULT, H5P_DEFAULT);
if (data_id<0) cgp_doError;

// Buffer to hold that data array
int data[2];

// Fill the buffer [Order needs verification]
data[0] = cell_dim;
data[1] = phys_dim;

// Write the data array to the file
herr = H5Dwrite(data_id, H5T_NATIVE_INT, shape_id, shape_id, H5P_DEFAULT, data);
if (herr<0) cgp_doError;

// Close the data array
herr = H5Dclose(data_id);
if (herr<0) cgp_doError;

// Destroy the shape object
herr = H5Sclose(shape_id);
if (herr<0) cgp_doError;

return 0;

int cgp_nbases(int fn, int *nbases) {

}
// Read the number of bases in this file
*nbases = files[fn].nbases;
return 0;

//= Zone IO Prototypes =/
int cgp_zone_read(int fn, int B, int Z, char* zonename, int* nijk) {
    if(fn>=files_count||fn<0) cgp_doError;
    if(!files[fn].isOpen) cgp_doError;
    if(B>files[fn].nbases||B<=0) cgp_doError;
    if(Z>files[fn].bases[B-1].nzones||Z<=0) cgp_doError;
    herr_t herr;
    int err;
    // Pointer to the current base
    base_t* base = &(files[fn].bases[B-1]);
    // Pointer to the zone to be read
    zone_t* zone = &(files[fn].bases[B-1].zones[Z-1]);
    // Copy the zone’s name
    strcpy(zonename, zone->zonename);
    // Loop index
    int k;
    // If the zone already has a coordinates sub-group, read it
    if(node_exists(zone->group_id, "GridCoordinates")) {
        // Test if the existing HDF5 handle for the grid_id is valid
        // If it is, use it, otherwise open a new handle
        if(!H5Iis_valid(zone->grid_id)) zone->grid_id = H5Gopen2(zone->group_id, "GridCoordinates", H5P_DEFAULT);
        if(zone->grid_id<0) cgp_doError;
        // Count the number of data arrays in the group
        num_nodes(zone->grid_id, "DataArray_t", &zone->ncoords);
        // Allocate memory to describe the data arrays
        zone->coords = (coords_t*) malloc(zone->ncoords*sizeof(coords_t));
        if(zone->coords==NULL) cgp_doError;
        // For each ncoord in ncoords
        for(k=0;k<zone->ncoords;k++) {
            // Do something with zone->coords[k]
// Buffer for the name of the coords
char coordname[100+1];
// Read the name of the coords
node_idx2name(zone->grid_id, "DataArray_t", k, coordname);
// Open the data array
zone->coords[k].group_id = H5Gopen2(zone->grid_id, coordname, H5P_DEFAULT);
if(zone->coords[k].group_id<0) cgp_doError;
// Set the index of the data
zone->coords[k].idx = k;
// Copy the name of the coords
strcpy(zone->coords[k].coordname,coordname);
}
else {
    // Create a new group to hold the coords
    new_node(zone->group_id, "GridCoordinates", "GridCoordinates_t", "MT");
    // Open the new group
    zone->grid_id = H5Gopen2(zone->group_id, "GridCoordinates", H5P_DEFAULT);
    if(zone->grid_id<0) cgp_doError;
    // Default to hold no coords data
    zone->ncoords = 0;
    zone->coords = NULL;
}
else {
    // If the zone already has a flow sub-group, read it
    if(node_exists(zone->group_id, "GridCoordinates")) {
        // Test if the existing HDF5 handle for the flow_id is valid
        // If it is, use it, otherwise open a new handle
        if(!H5Iis_valid(zone->flow_id)) zone->flow_id = H5Gopen2(zone->flow_id, "FlowSolution", H5P_DEFAULT);
        if(zone->flow_id<0) cgp_doError;
        // Count the number of data arrays in the group
        num_nodes(zone->flow_id, "DataArray_t", &(zone->ncoords));
// Allocate memory to describe the data arrays
zone->sols = (sol_t *) malloc(zone->nsols * sizeof(sol_t));
if (zone->sols == NULL) cgp_doError;
// For each sol in nsols
for (k = 0; k < zone->nsols; k++) {
    // Buffer for the name of the coords
    char solname[100 + 1];
    // Read the name of the coords
    node_idx2name(zone->flow_id, "DataArray_t", k, solname);
    // Open the data array
    zone->sols[k].group_id = H5Gopen2(zone->flow_id, solname, H5P_DEFAULT);
    if (zone->sols[k].group_id < 0) cgp_doError;
    // Set the index of the data
    zone->sols[k].idx = k;
    // Copy the name of the coords
    strcpy(zone->sols[k].solname, solname);
}
// If the zone does not have a flow sub-group, create one for it
else {
    // Create a new group to hold the coords
    new_node(zone->group_id, "FlowSolution", "FlowSolution_t", "MT");
    // Open the new group
    zone->flow_id = H5Gopen2(zone->group_id, "FlowSolution", H5P_DEFAULT);
    if (zone->flow_id < 0) cgp_doError;
    // Default to hold no coords data
    zone->nsols = 0;
    zone->sols = NULL;
}
// Count the number of element sections
num_nodes(zone->group_id, "Elements_t", &zone->nsections);
// Allocate memory to describe the element sections
zone->sections = (section_t *) malloc(zone->nsections * sizeof(section_t));
// For each section in nsections
for (k=0; k<zone->nsections; k++) {
    // Buffer for the section’s name
    char sectionname[100+1];
    // Read the name of the k’th section
    node_id2name(zone->group_id, "Elements_t", k, sectionname);
    // Open the section
    zone->sections[k].group_id = H5Gopen2(zone->group_id, sectionname, H5P_DEFAULT);
    if (zone->sections[k].group_id<0) cgp_doError;
    // Open the section’s connectivity
    zone->sections[k].connectivity_id = H5Gopen2(zone->sections[k].group_id, "ElementConnectivity", H5P_DEFAULT);
    if (zone->sections[k].connectivity_id<0) cgp_doError;
    // Open the section’s ranges
    zone->sections[k].range_id = H5Gopen2(zone->sections[k].group_id, "ElementRange", H5P_DEFAULT);
    if (zone->sections[k].range_id<0) cgp_doError;
    // Set the index of the section
    zone->sections[k].idx = k;
    // Copy the name of the section
    strcpy(zone->sections[k].sectionname, sectionname);
    // Read the element type
    {
        // Dimensions of data
        hsize_t dims[1] = {2};
        // Shape of data in memory and the file
        hid_t shape_id = H5Screate_simple(1, dims, NULL);
        // Open the array
        hid_t data_id = H5Dopen2(zone->sections[k].group_id, "data", H5P_DEFAULT);
        // Buffer to hold results
        int data[2];
        // Read array
        H5Dread(data_id, H5T_NATIVE_INT, shape_id, shape_id, H5P_DEFAULT, data);
        // Set the type
zone->sections[k].type = data[0];

// Close the array
H5Dclose(data_id);

// Close the shape
H5Sclose(shape_id);

// Read the element range
{
    // Dimensions of data
    hsize_t dims[1] = {2};

    // Shape of data in memory and the file
    hid_t shape_id = H5Screate_simple(1,dims,NULL);

    // Open the array
    hid_t data_id = H5Dopen2(zone->sections[k].range_id, "data", H5P_DEFAULT);

    // Buffer to hold results
    int data[2];

    // Read array
    H5Dread(data_id,H5T_NATIVE_INT, shape_id,shape_id,H5P_DEFAULT,data);

    // Set the range
    zone->sections[k].range[0] = data[0];
    zone->sections[k].range[1] = data[1];

    // Close the array
    H5Dclose(data_id);

    // Close the shape
    H5Sclose(shape_id);

    }

// Count the number of flow solutions
num_nodes(zone->group_id, "FlowSolution_t", &zone->nsols);

// Allocate memory to describe the flow solutions
zone->sols = (sol_t*) malloc(zone->nsols*sizeof(sol_t));

if(zone->sols==NULL) cgp_doError;

// For each solution in nsols

for(k=0;k<zone->nsols;k++) {
    // Buffer for the solution's name
    char solname[100+1];
    // Read the name of the k'th solution
    node_idx2name(zone->group_id, "FlowSolution_t", k, solname);
    // Open the solution
    zone->sols[k].group_id = H5Gopen2(zone->group_id, solname, H5P_DEFAULT);
    if(zone->sols[k].group_id<0) cgp_doError;
    // Set the index of the solution
    zone->sols[k].idx = k;
    // Copy the name of the solution
    strcpy(zone->sols[k].solname,solname);
}

// Read the zone type
char ztype[100];
err = get_str(zone->group_id, "ZoneType/ data", 99, ztype);
if(strcmp(ztype,"Structured")) zone->type = Structured;
else if(strcmp(ztype,"Unstructured")) zone->type = Unstructured;
else cgp_doError;

// Read nijk
// Set the size of the array to read
int cols = (zone->type==Structured)?base->cell_dim:1;
hsize_t dim[2] = {3, cols};
// Create the shape object for the array
hid_t shape_id = H5Screate_simple(2,dim, NULL);
if(shape_id<0) cgp_doError;
// Open the array in the file
hid_t data_id = H5Dopen2(zone->group_id, " data", H5P_DEFAULT);
if(data_id<0) cgp_doError;
// Read the array from the file to the local nijk
herr = H5Dread(data_id, H5T_NATIVE_INT, shape_id, shape_id, H5P_DEFAULT, nijk);
if(herr<0) cgp_doError;
// Allocate space to store the data
zone->nijk = malloc(3*cols*sizeof(int));
if(nijk==NULL) cgp_doError;

// Read the array from the file to the global nijk
herr = H5Dread(data_id, H5T_NATIVE_INT, shape_id, shape_id, H5P_DEFAULT, zone->nijk);
if(herr<0) cgp_doError;

// Close the array
herr = H5Dclose(data_id);
if(herr<0) cgp_doError;

// Destroy the shape
herr = H5Sclose(shape_id);
if(herr<0) cgp_doError;
return 0;

int cgp_zone_type(int fn, int B, int Z, ZoneType_t *zonetype) {
    // Read Zonetype from file and return in enum
    file_t* file = &(files[fn]);
    base_t* base = &(file->bases[B-1]);
    zone_t* zone = &(base->zones[Z-1]);
    *zonetype = zone->type;
    return 0;
}

int cgp_zone_write(int fn, int B, const char* zonename, const int* nijk, ZoneType_t type, int* Z) {
    if(fn>=files_count||fn<0) cgp_doError;
    if(!files[fn].isOpen) cgp_doError;
    if(B>files[fn].nbases||B<=0) cgp_doError;
    herr_t herr;
    // Pointer to the base
    base_t* base = &(files[fn].bases[B-1]);
    // Index
    int idx;
// Loop index
int k;

// If the zone exists, replace it
if(node_exists(base->group_id, zonename)) {
    // Find the zone in this base
    for(k=0;k<base->nzones;k++)
        if(strcmp(zonename,base->zones[k].zonename)==0)
            idx = k;

    // Free the memory for the zone
    free_zone(&(base->zones[idx]));

    // Delete the zone from the file
    del_node(base->group_id,zonename);
}

// If the zone does not exist, initialize memory for it
else {
    // Allocate a bigger list of zones
    zone_t* zones = (zone_t*) malloc((base->nzones+1)*sizeof(zone_t));
    if(zones==NULL) cgp_doError;

    // Copy the old zones to the new list
    for(k=0;k<base->nzones;k++)
        zones[k] = base->zones[k];

    // Deallocate the old list
    if(base->zones!=NULL) free(base->zones);

    // Point the base’s list to the new list
    base->zones = zones;

    // Increment the number of zones
    base->nzones++;

    // Set the index of the zone
    idx = base->nzones-1;
}

// Pointer to the current zone
zone_t* zone = &(base->zones[idx]);

// Copy the name of the zone
strcpy(zone->zonename, zonename);

// Set the index of the zone
zone->idx = idx;

// Default to no coods or solutions
zone->coords = NULL;
zone->ncoords = 0;
zone->sections = NULL;
zone->nsections = 0;
zone->sols = NULL;
zone->nsols = 0;

// Return the index of the zone
*Z = idx+1;

// Create the zone in the file
new_node(base->group_id, zonename, "Zone_t", "I4");

// Open the new zone
zone->group_id = H5Gopen2(base->group_id, zonename, H5P_DEFAULT);
if(zone->group_id<0) cgp_doError;

// Write ZoneType
zone->type = type;

// Create a new node for the zone type
new_node(zone->group_id, "ZoneType", "ZoneType_t", "C1");

// Open the new node
hid_t group_id = H5Gopen2(zone->group_id, "ZoneType", H5P_DEFAULT);
if(group_id<0) cgp_doError;

// Write a string as the data for the new node
if(type==Structured) new_str(group_id, " data", "Structured", strlen("Structured")-1);
else if(type==Unstructured) new_str(group_id, " data", "Unstructured", strlen("Unstructured")-1);
else cgp_doError;

// Close the node
herr = H5Gclose(group_id);
if(herr<0) cgp_doError;

// Write nijk
// Size of array to write
int cols = (type==Structured)?base->cell_dim:1;
hsizes_t dim[2] = {3, cols};

// Create shape object for array to write
hid_t shape_id = H5Screate_simple(2, dim, NULL);
if (shape_id<0) cgp_doError;

// Create the data array in the file
hid_t data_id = H5Dcreate2(zone->group_id, "data", H5T_NATIVE_INT, shape_id, H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
if (data_id<0) cgp_doError;

// Write the data to the array
herr = H5Dwrite(data_id, H5T_NATIVE_INT, shape_id, shape_id, H5P_DEFAULT, nijk);
if (herr<0) cgp_doError;

// Allocate space to store the data
zone->nijk = malloc(3*cols*sizeof(int));
if (zone->nijk==NULL) cgp_doError;

// Read the data to the zone in memory
herr = H5Dread(data_id, H5T_NATIVE_INT, shape_id, shape_id, H5P_DEFAULT, zone->nijk);
if (herr<0) cgp_doError;

// Close the data array
herr = H5Dclose(data_id);
if (herr<0) cgp_doError;

// Destroy the shape object
herr = H5Sclose(shape_id);
if (herr<0) cgp_doError;

// Write GridCoordinates
new_node(zone->group_id, "GridCoordinates", "GridCoordinates_t", "MT");

// Open the node
zone->grid_id = H5Gopen2(zone->group_id, "GridCoordinates", H5P_DEFAULT);
if (zone->grid_id<0) cgp_doError;

// Write FlowSolution
new_node(zone->group_id, "FlowSolution", "FlowSolution_t", "MT");
new_node(zone->group_id, "FlowSolution", "FlowSolution_t", "MT");

// Open the node
zone->flow_id = H5Gopen2(zone->group_id, "FlowSolution", H5P_DEFAULT);
if(zone->grid_id<0) cgp_doError;
return 0;
}

int cgp_nzones(int fn, int B, int *nzones) {
    // Read the number of zones in this base
    *nzones = files[fn].bases[B-1].nzones;
    return 0;
}

//= Grid IO Prototypes =//
int cgp_coord_write(int fn, int B, int Z, DataType_t type, const char* coordname, int* C) {
    if(fn>=files_count||fn<0) cgp_doError;
    if(!files[fn].isOpen) cgp_doError;
    if(B>files[fn].nbases||B<=0) cgp_doError;
    if(Z>files[fn].bases[B-1].nzones||Z<=0) cgp_doError;
    herr_t herr;
    // Pointer to the current file
    file_t* file = &(files[fn]);
    // Pointer to the current base
    base_t* base = &(file->bases[B-1]);
    // Pointer to the current zone
    zone_t* zone = &(base->zones[Z-1]);
    // Index
    int idx;
    // Loop index
    int k;
    // If the coordinates already exist, replace them
    if(node_exists(zone->grid_id, coordname)) {
        // Find the coordinates to be replaced
for(k=0;k<zone->ncoords;k++) if(strcmp(coordname,zone->coords[k].coordname) ==0) idx = k;

// Delete them
del_node(zone->grid_id,coordname);

// If the coordinates do not exist, prepare the memory structures
else {
    // New list of coords
    coords_t* coords = (coords_t*) malloc((zone->ncoords+1)*sizeof(coords_t));
    if(coords==NULL) cgp_doError;
    // Copy old list to new list
    for(k=0;k<zone->ncoords;k++) coords[k] = zone->coords[k];
    // Free the old list
    if(zone->coords!=NULL) free(zone->coords);
    // Point the zone to the new list
    zone->coords = coords;
    // Incement the number of coords
    zone->ncoords++;
    // Set the coords index
    idx = zone->ncoords-1;
}

// Pointer to coords
coords_t* coords = &(zone->coords[idx]);

// Copy the name
strcpy(coords->coordname, coordname);

// Set the index
coords->idx = idx;

// Return the index
*C = idx+1;

// Create a new node in the file
new_node(zone->grid_id, coordname, "DataArray_t", "R8");

// Open the node
coords->group_id = H5Gopen2(zone->grid_id, coordname, H5P_DEFAULT);
if(coords->group_id<0) cgp_doError;
// Set the rank of the dimensions
hsize_t rank = (zone->type==Structured)?base->cell_dim:1;

// Dimensions of the array
hsize_t dims[rank];

// Set these to correspond with the size of the zone
for(k=0;k<rank;k++) dims[k] = zone->nijk[(rank-1)-k];

// Create a shape for the array
hid_t shape_id = H5Screate_simple(rank,dims,NULL);

if(shape_id<0) cgp_doError;

// Property list for dataset
hid_t plist_id = H5Pcreate(H5P_DATASET_CREATE);

if(preallocate) herr = H5Pset_alloc_time(plist_id,H5D_ALLOC_TIME_EARLY);

if(preallocate) herr = H5Pset_fill_time(plist_id, H5D.FILL_TIME_ALLOC);

// Create the array in the file
hid_t data_id = H5Dcreate2(coords->group_id, " data", H5T_NATIVE_DOUBLE, shape_id, H5P_DEFAULT, plist_id, H5P_DEFAULT);

if(data_id<0) cgp_doError;

// Close the array
herr = H5Dclose(data_id);

if(herr<0) cgp_doError;

// Close the property list
herr = H5Pclose(plist_id);

if(herr<0) cgp_doError;

// Close the shape
herr = H5Sclose(shape_id);

if(herr<0) cgp_doError;

return 0;

int cgp_coord_write_data(int fn, int B, int Z, int C, int* min, int* max, void* coord_array) {
    int k;

    if fn>=files_count || fn<0) cgp_doError;
if(!files[fn].isOpen) cgp_doError;

if(B>files[fn].nbases||B<=0) cgp_doError;

if(Z>files[fn].bases[B-1].nzones||Z<=0) cgp_doError;

if(C>files[fn].bases[B-1].zones[Z-1].ncoords||C<=0) cgp_doError;

herr_t herr;

// Pointer to the current file
file_t* file = &(files[fn]);

// Pointer to the current base
base_t* base = &(file->bases[B-1]);

// Pointer to the current zone
zone_t* zone = &(base->zones[Z-1]);

// Pointer to the current coords
coords_t* coords = &(zone->coords[C-1]);

// Open the data
hid_t data_id = H5Dopen2(coords->group_id, "data", H5P_DEFAULT);
if(data_id<0) cgp_doError;

// Set the rank of the data
hsize_t rank = (zone->type==Structured)?base->cell_dim:1;

// Set the start position for the data write
hsize_t start[rank];

for(k=0;k<rank;k++) start[k] = min[k];

// Compute the counts in each dimension
hsize_t dims[rank];

for(k=0;k<rank;k++) dims[k] = max[k]-min[k]+1;

// Create a shape for the data in memory
hid_t mem_shape_id = H5Screate_simple(rank,dims,NULL);
if(mem_shape_id<0) cgp_doError;

// Create a shape for the data in the file
hid_t data_shape_id = H5Dget_space(data_id);
if(data_shape_id<0) cgp_doError;

// Select a section of the array in the file
herr = H5Sselect_hyperslab(data_shape_id, H5S_SELECT_SET, start, NULL, dims, NULL);
210

if(herr<0) cgp_doError;

// Set the access property list for data transfer
hid_t plist_id = H5Pcreate(H5P_DATASET_XFER);
if(plist_id<0) cgp_doError;

// Set MPI-I/O collective communication
herr = H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);
if(herr<0) cgp_doError;

// Write the data in collective parallel I/O
herr = H5Dwrite(data_id, H5T_NATIVE_DOUBLE, mem_shape_id, data_shape_id, plist_id, coord_array);
if(herr<0) cgp_doError;

// Close the property list
herr = H5Pclose(plist_id);
if(herr<0) cgp_doError;

// Close the shape of the data in the file
herr = H5Sclose(data_shape_id);
if(herr<0) cgp_doError;

// Close the shape of the data in memory
herr = H5Sclose(mem_shape_id);
if(herr<0) cgp_doError;

// Close the data array
herr = H5Dclose(data_id);
if(herr<0) cgp_doError;

return 0;

int cgp_sol_write_data(int fn, int B, int Z, int S, int* min, int* max, void* data) {
  int k;

  if(fn>=files_count||fn<0) cgp_doError;
  if(!files[fn].isOpen) cgp_doError;
  if(B>files[fn].nbases||B<=0) cgp_doError;
  if(Z>files[fn].bases[B-1].nzones||Z<=0) cgp_doError;
  if(S>files[fn].bases[B-1].zones[Z-1].nsols||S<=0) cgp_doError;
herr_t herr;

// Pointer to the current file
file_t* file = &(files[fn]);

// Pointer to the current base
base_t* base = &(file->bases[B-1]);

// Pointer to the current zone
zone_t* zone = &(base->zones[Z-1]);

// Pointer to the current coords
sol_t* sol = &(zone->sols[S-1]);

// Open the data
hid_t data_id = H5Dopen2(sol->group_id, "data", H5P_DEFAULT);
if(data_id<0) cgp_doError;

// Set the rank of the data
hsize_t rank = (zone->type==Structured)?base->cell_dim:1;

// Set the start position for the data write
hsize_t start[rank];

for(k=0;k<rank;k++) start[k] = min[k];

// Compute the counts in each dimension
hsize_t dims[rank];

for(k=0;k<rank;k++) dims[k] = max[k]-min[k]+1;

// Create a shape for the data in memory
hid_t mem_shape_id = H5Screate_simple(rank,dims,NULL);
if(mem_shape_id<0) cgp_doError;

// Create a shape for the data in the file
hid_t data_shape_id = H5Dget_space(data_id);
if(data_shape_id<0) cgp_doError;

// Select a section of the array in the file
herr = H5Sselect_hyperslab(data_shape_id, H5S_SELECT_SET, start, NULL, dims, NULL);
if(herr<0) cgp_doError;

// Set the access property list for data transfer
hid_t plist_id = H5Pcreate(H5P_DATASET_XFER);
if(plist_id<0) cgp_doError;
00854  // Set MPI-IO collective communication
00855  herr = H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);
00856  if(herr<0) cgp_doError;
00857  // Write the data in collective parallel I/O
00858  herr = H5Dwrite(data_id, H5T_NATIVE_DOUBLE, mem_shape_id, data_shape_id, plist_id, data);
00859  if(herr<0) cgp_doError;
00860  // Close the property list
00861  herr = H5Pclose(plist_id);
00862  if(herr<0) cgp_doError;
00863  // Close the shape of the data in the file
00864  herr = H5Sclose(data_shape_id);
00865  if(herr<0) cgp_doError;
00866  // Close the shape of the data in memory
00867  herr = H5Sclose(mem_shape_id);
00868  if(herr<0) cgp_doError;
00869  // Close the data array
00870  herr = H5Dclose(data_id);
00871  if(herr<0) cgp_doError;
00872
00873  return 0;
00874 }
00875
00876  int cgp_sol_write(int fn, int B, int Z, char *solname, GridLocation_t location, int *S) {
00877    if(fn>=files_count||fn<0) cgp_doError;
00878    if(!files[fn].isOpen) cgp_doError;
00879    if(B>files[fn].nbases||B<=0) cgp_doError;
00880    if(Z>files[fn].bases[B-1].nzones||Z<=0) cgp_doError;
00881    herr_t herr;
00882    // Pointer to the current file
00883    file_t* file = &(files[fn]);
00884    // Pointer to the current base
00885    base_t* base = &(file->bases[B-1]);
00886    // Pointer to the current zone
zone_t* zone = &(base->zones[Z-1]);

// Index
int idx;
// Loop index
int k;

// If the solution already exist, replace it
if(node_exists(zone->flow_id, solname)) {
    // Find the coordinates to be replaced
    for(k=0;k<zone->nsols;k++) if(strcmp(solname,zone->sols[k].solname)==0) idx = k;
    // Delete them
    del_node(zone->flow_id,solname);
}

// If the coordinates do not exist, prepare the memory structures
else {
    // New list of coords
    sol_t* sols = (sol_t*) malloc((zone->nsols+1)*sizeof(sol_t));
    if(sols==NULL) cgp_doError;
    // Copy old list to new list
    for(k=0;k<zone->nsols;k++) sols[k] = zone->sols[k];
    // Free the old list
    if(zone->sols!=NULL) free(zone->sols);
    // Point the zone to the new list
    zone->sols = sols;
    // Increment the number of coords
    zone->nsols++;
    // Set the coords index
    idx = zone->nsols-1;
}

// Pointer to coords
sol_t* sol = &(zone->sols[idx]);
// Copy the name
strcpy(sol->solname, solname);
// Set the index
sol->idx = idx;
// Return the index
*S = idx+1;

// Create a new node in the file
new_node(zone->flow_id, solname, "DataArray_t", "R8");

// Open the node
sol->group_id = H5Gopen2(zone->flow_id, solname, H5P_DEFAULT);
if(sol->group_id<0) cgp_doError;

// Set the rank of the dimensions
hsize_t rank = (zone->type==Structured)?base->cell_dim:1;

// Dimensions of the array
hsize_t dims[rank];

// Set these to correspond with the size of the zone
if(location==Vertex) for(k=0;k<rank;k++) dims[k] = zone->nijk[(rank-1)-k];
else if(location==CellCenter) for(k=0;k<rank;k++) dims[k] = zone->nijk[rank+(rank-1)-k];

// Create a shape for the array
hid_t shape_id = H5Screate_simple(rank,dims,NULL);
if(shape_id<0) cgp_doError;

// Property list for dataset
hid_t plist_id = H5Pcreate(H5P_DATASET_CREATE);
if(preallocate) herr = H5Pset_alloc_time(plist_id,H5D_ALLOC_TIME_EARLY);
if(preallocate) herr = H5Pset_fill_time(plist_id, H5D_FILL_TIME_ALLOC);

// Create the array in the file
hid_t data_id = H5Dcreate2(sol->group_id, " data", H5T_NATIVE_DOUBLE, shape_id , H5P_DEFAULT, plist_id, H5P_DEFAULT);
if(data_id<0) cgp_doError;

// Close the array
herr = H5Dclose(data_id);
if(herr<0) cgp_doError;

// Close the property list
herr = H5Pclose(plist_id);
if(herr<0) cgp_doError;
00946  // Close the shape
00947  herr = H5Sclose(shape_id);
00948  if(herr<0) cgp_doError;
00949  return 0;
00950 }

00951 int cgp_nsols(int fn, int B, int Z, int* nsols) {
00952  if(fn>=files_count||fn<0) cgp_doError;
00953  if(!files[fn].isOpen) cgp_doError;
00954  if(B>files[fn].nbases||B<=0) cgp_doError;
00955  if(Z>files[fn].bases[B-1].nzones||Z<=0) cgp_doError;
00956  herr_t herr;
00957  // Pointer to the current file
00958  file_t* file = &(files[fn]);
00959  // Pointer to the current base
00960  base_t* base = &(file->bases[B-1]);
00961  // Pointer to the current zone
00962  zone_t* zone = &(base->zones[Z-1]);
00963  *nsols = zone->nsols;
00964  return 0;
00965 }

00966 int cgp_section_write(int fn, int B, int Z, char* sectionname, ElementType_t type,
00967  int start, int end, int nbdry, int* S) {
00968  if(fn>=files_count||fn<0) cgp_doError;
00969  if(!files[fn].isOpen) cgp_doError;
00970  if(B>files[fn].nbases||B<=0) cgp_doError;
00971  if(Z>files[fn].bases[B-1].nzones||Z<=0) cgp_doError;
00972  herr_t herr;
00973  // Pointer to the current file
00974  file_t* file = &(files[fn]);
00975  // Pointer to the current base
00976  base_t* base = &(file->bases[B-1]);
// Pointer to the current zone
zone_t* zone = &(base->zones[Z-1]);

// Index
int idx;

// Loop index
int k;

// If the section already exist, replace it
if(node_exists(zone->grid_id, sectionname)) {
  // Find the section to be replaced
  for(k=0;k<zone->nsections;k++)
      if(strcmp(sectionname,zone->sections[k].sectionname)==0) idx = k;

  // Delete it
  del_node(zone->grid_id,sectionname);
}

// If the section does not exist, prepare the memory structures
else {
  // New list of sections
  section_t* sections = (section_t*) malloc((zone->nsections+1)*sizeof(section_t));

  if(sections==NULL) cgp_doError;

  // Copy old list to new list
  for(k=0;k<zone->nsections;k++) sections[k] = zone->sections[k];

  // Free the old list
  if(zone->sections!=NULL) free(zone->sections);

  // Point the zone to the new list
  zone->sections = sections;

  // Increment the number of coords
  zone->nsections++;

  // Set the coords index
  idx = zone->nsections-1;
}

// Pointer to coords
section_t* section = &(zone->sections[idx]);
// Copy the name
strcpy(section->sectionname, sectionname);

// Set the index
section->idx = idx;

// Set the type
section->type = type;

// Set the range
section->range[0] = start;
section->range[1] = end;

// Return the index
*S = idx+1;

// Create the section in the file
new_node(zone->group_id, sectionname, "Elements_t", "I4");

// Open the new zone
section->group_id = H5Gopen2(zone->group_id, sectionname, H5P_DEFAULT);
if(section->group_id<0) cgp_doError;

// Write data
{
    // Size of array to write
    hsize_t dim[1] = {2};
    // Create shape object for array to write
    hid_t shape_id = H5Screate_simple(1, dim, NULL);
    if(shape_id<0) cgp_doError;
    // Create the data array in the file
    hid_t data_id = H5Dcreate2(section->group_id, " data", H5T_NATIVE_INT, shape_id, H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
    if(data_id<0) cgp_doError;
    // Write the data to the array
    int data[2] = {type,0};
    herr = H5Dwrite(data_id, H5T_NATIVE_INT, shape_id, shape_id, H5P_DEFAULT, data);
    if(herr<0) cgp_doError;
    // Close the data array
01054  herr = H5Dclose(data_id);
01055  if(herr<0) cgp_doError;
01056  // Destroy the shape object
01057  herr = H5Sclose(shape_id);
01058  if(herr<0) cgp_doError;
01059 }
01060
01061  // Create the ElementConnectivity in the section
01062  new_node(section->group_id, "ElementConnectivity", "DataArray_t", "I4");
01063  // Open the new zone
01064  section->connectivity_id = H5Gopen2(section->group_id, "ElementConnectivity",
H5P_DEFAULT);
01065  if(section->connectivity_id<0) cgp_doError;
01066
01067  // Create array for data
01068  // We choose not to support the MIXED type
01069  // The array size is (number of elements)*(nodes per element)
01070  // Write data
01071{
01072   // Size of array to write
01073  hsize_t dim[1] = {(end-start+1)*node_counts[type]};
01074  // Create shape object for array to write
01075  hid_t shape_id = H5Screate_simple(1,dim, NULL);
01076  if(shape_id<0) cgp_doError;
01077  // Property list for dataset
01078  hid_t plist_id = H5Pcreate(H5P_DATASET_CREATE);
01079  if(preallocate) herr = H5Pset_alloc_time(plist_id,H5D_ALLOC_TIME_EARLY);
01080  if(preallocate) herr = H5Pset_fill_time(plist_id, H5D_FILL_TIME_ALLOC);
01081  // Create the data array in the file
01082  hid_t data_id = H5Dcreate2(section->connectivity_id, " data", H5T_NATIVE_INT,
shape_id, H5P_DEFAULT, plist_id, H5P_DEFAULT);
01083  if(data_id<0) cgp_doError;
01084  // Close the data array
01085  herr = H5Dclose(data_id);
01086  if(herr<0) cgp_doError;
// Close the property list
herr = H5Pclose(plist_id);
if(herr<0) cgp_doError;

// Destroy the shape object
herr = H5Sclose(shape_id);
if(herr<0) cgp_doError;
}

// Create the ElementRange in the section
new_node(section->group_id, "ElementRange", "IndexRange_t", "I4");

// Open the new zone
section->range_id = H5Gopen2(section->group_id, "ElementRange", H5P_DEFAULT);
if(section->range_id<0) cgp_doError;

// Write data
{
    // Size of array to write
hsize_t dim[1] = {2};

    // Create shape object for array to write
hid_t shape_id = H5Screate_simple(1, dim, NULL);
if(shape_id<0) cgp_doError;

    // Create the data array in the file
hid_t data_id = H5Dcreate2(section->range_id, " data", H5T_NATIVE_INT, shape_id, H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
if(data_id<0) cgp_doError;

    // Write the data to the array
int data[2] = {start, end};
herr = H5Dwrite(data_id, H5T_NATIVE_INT, shape_id, shape_id, H5P_DEFAULT, data);
if(herr<0) cgp_doError;

    // Close the data array
herr = H5Dclose(data_id);
if(herr<0) cgp_doError;

    // Destroy the shape object
herr = H5Sclose(shape_id);
int cgp_section_write_data(int fn, int B, int Z, int S, int min, int max, int *elements) {
    int k;
    if(fn>=files_count||fn<0) cgp_doError;
    if(!files[fn].isOpen) cgp_doError;
    if(B>files[fn].nbases||B<=0) cgp_doError;
    if(Z>files[fn].bases[B-1].nzones||Z<=0) cgp_doError;
    if(S>files[fn].bases[B-1].zones[Z-1].nsections||S<=0) cgp_doError;
    herr_t herr;
    // Open the data
    hid_t data_id = H5Dopen2(section->connectivity_id, "data", H5P_DEFAULT);
    if(data_id<0) cgp_doError;
    // Set the rank of the data
    hsize_t rank = 1;
    // Set the start position for the data write
    hsize_t start[rank];
    start[0] = (min-section->range[0])*node_counts[section->type];
    // Compute the counts in each dimension
    hsize_t dims[rank];
dims[0] = (max-min+1) * node_counts[section->type];

// Create a shape for the data in memory
hid_t mem_shape_id = H5Screate_simple(rank, dims, NULL);
if (mem_shape_id < 0) cgp_doError;

// Create a shape for the data in the file
hid_t data_shape_id = H5Dget_space(data_id);
if (data_shape_id < 0) cgp_doError;

// Select a section of the array in the file
herr = H5Sselect_hyperslab(data_shape_id, H5S_SELECT_SET, start, NULL, dims, NULL);
if (herr < 0) cgp_doError;

// Set the access property list for data transfer
hid_t plist_id = H5Pcreate(H5P_DATASET_XFER);
if (plist_id < 0) cgp_doError;

// Set MPI-I0 collective communication
herr = H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);
if (herr < 0) cgp_doError;

// Write the data in collective parallel I/O
herr = H5Dwrite(data_id, H5T_NATIVE_INT, mem_shape_id, data_shape_id, plist_id, elements);
if (herr < 0) cgp_doError;

// Close the property list
herr = H5Pclose(plist_id);
if (herr < 0) cgp_doError;

// Close the shape of the data in the file
herr = H5Sclose(data_shape_id);
if (herr < 0) cgp_doError;

// Close the shape of the data in memory
herr = H5Sclose(mem_shape_id);
if (herr < 0) cgp_doError;

// Close the data array
herr = H5Dclose(data_id);
if (herr < 0) cgp_doError;

return 0;
int cgp_array_write(int fn, int B, int Z, char *arrayname, GridLocation_t location) {
    if (fn >= files_count || fn < 0) cgp_doError;
    if (!files[fn].isOpen) cgp_doError;
    if (B > files[fn].nbases || B <= 0) cgp_doError;
    if (Z > files[fn].bases[B-1].nzones || Z <= 0) cgp_doError;
    herr_t herr;
    // Pointer to the current file
    file_t* file = &(files[fn]);
    // Pointer to the current base
    base_t* base = &(file->bases[B-1]);
    // Pointer to the current zone
    zone_t* zone = &(base->zones[Z-1]);
    // Index
    int idx;
    // Loop index
    int k;
    // If the solution already exist, replace it
    if (node_exists(zone->group_id, arrayname)) {
        // Delete them
        del_node(zone->group_id, arrayname);
    }
    // Create a new node in the file
    new_node(zone->group_id, arrayname, "DataArray_t", "R8");
    // Open the node
    hid_t group_id = H5Gopen2(zone->group_id, arrayname, H5P_DEFAULT);
    if (group_id < 0) cgp_doError;
    // Set the rank of the dimensions
    hsize_t rank = (zone->type == Structured) ? base->cell_dim : 1;
// Dimensions of the array
hsiz_t dims[rank];

// Set these to correspond with the size of the zone
if (location==Vertex) for(k=0;k<rank;k++) dims[k] = zone->nijk[(rank-1)-k];
else if (location==CellCenter) for(k=0;k<rank;k++) dims[k] = zone->nijk[rank+(rank-1)-k];

// Create a shape for the array
hid_t shape_id = H5Screate_simple(rank,dims,NULL);
if (shape_id<0) cgp_doError;

// Property list for dataset
hid_t plist_id = H5Pcreate(H5P_DATASET_CREATE);
if (!preallocate) herr = H5Pset_alloc_time(plist_id,H5D_ALLOC_TIME_EARLY);
if (!preallocate) herr = H5Pset_fill_time(plist_id, H5D_FILL_TIME_ALLOC);

// Create the array in the file
hid_t data_id = H5Dcreate2(group_id, "data", H5T_NATIVE_DOUBLE, shape_id, H5P_DEFAULT, plist_id, H5P_DEFAULT);
if (data_id<0) cgp_doError;

// Close the array
herr = H5Dclose(data_id);

// Close the property list
herr = H5Pclose(plist_id);
if (herr<0) cgp_doError;

// Close the shape
herr = H5Sclose(shape_id);
if (herr<0) cgp_doError;

// Close the group
herr = H5Gclose(group_id);
if (herr<0) cgp_doError;
return 0;
}

cgp_array_write_data(int fn, int B, int Z, char* arrayname, int* min, int* max, void* data) {
  int k;
  // Dimensions of the array
  hsize_t dims[rank];
  // Set these to correspond with the size of the zone
  if (location==Vertex) for(k=0;k<rank;k++) dims[k] = zone->nijk[(rank-1)-k];
else if (location==CellCenter) for(k=0;k<rank;k++) dims[k] = zone->nijk[rank+(rank-1)-k];

  // Create a shape for the array
  hid_t shape_id = H5Screate_simple(rank,dims,NULL);
  if (shape_id<0) cgp_doError;

  // Property list for dataset
  hid_t plist_id = H5Pcreate(H5P_DATASET_CREATE);
  if (!preallocate) herr = H5Pset_alloc_time(plist_id,H5D_ALLOC_TIME_EARLY);
  if (!preallocate) herr = H5Pset_fill_time(plist_id, H5D_FILL_TIME_ALLOC);

  // Create the array in the file
  hid_t data_id = H5Dcreate2(group_id, "data", H5T_NATIVE_DOUBLE, shape_id, H5P_DEFAULT, plist_id, H5P_DEFAULT);
  if (data_id<0) cgp_doError;

  // Close the array
  herr = H5Dclose(data_id);

  // Close the property list
  herr = H5Pclose(plist_id);
  if (herr<0) cgp_doError;

  // Close the shape
  herr = H5Sclose(shape_id);
  if (herr<0) cgp_doError;

  // Close the group
  herr = H5Gclose(group_id);
  if (herr<0) cgp_doError;
return 0;
if(fn>=files_count||fn<0) cgp_doError;
if(!files[fn].isOpen) cgp_doError;
if(B>files[fn].nbases||B<=0) cgp_doError;
if(Z>files[fn].bases[B-1].nzones||Z<=0) cgp_doError;
herr_t herr;

// Pointer to the current file
file_t* file = &(files[fn]);

// Pointer to the current base
base_t* base = &(file->bases[B-1]);

// Pointer to the current zone
zone_t* zone = &(base->zones[Z-1]);

// Open the group
hid_t group_id = H5Gopen2(zone->group_id, arrayname, H5P_DEFAULT);

// Open the data
hid_t data_id = H5Dopen2(group_id, " data", H5P_DEFAULT);
if(data_id<0) cgp_doError;

// Set the rank of the data
hsize_t rank = (zone->type==Structured)?base->cell_dim:1;

// Set the start position for the data write
hsize_t start[rank];
for(k=0;k<rank;k++) start[k] = min[k];

// Compute the counts in each dimension
hsize_t dims[rank];
for(k=0;k<rank;k++) dims[k] = max[k]-min[k]+1;

// Create a shape for the data in memory
hid_t mem_shape_id = H5Screate_simple(rank,dims,NULL);
if(mem_shape_id<0) cgp_doError;

// Create a shape for the data in the file
hid_t data_shape_id = H5Dget_space(data_id);
if(data_shape_id<0) cgp_doError;

// Select a section of the array in the file
herr = H5Sselect_hyperslab(data_shape_id, H5S_SELECT_SET, start, NULL, dims, NULL);
if(herr<0) cgp_doError;

// Set the access property list for data transfer
hid_t plist_id = H5Pcreate(H5P_DATASET_XFER);
if(plist_id<0) cgp_doError;

// Set MPI-IO collective communication
herr = H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);
if(herr<0) cgp_doError;

// Write the data in collective parallel I/O
herr = H5Dwrite(data_id, H5T_NATIVE_DOUBLE, mem_shape_id, data_shape_id, plist_id, data);
if(herr<0) cgp_doError;

// Close the property list
herr = H5Pclose(plist_id);
if(herr<0) cgp_doError;

// Close the shape of the data in the file
herr = H5Sclose(data_shape_id);
if(herr<0) cgp_doError;

// Close the shape of the data in memory
herr = H5Sclose(mem_shape_id);
if(herr<0) cgp_doError;

// Close the data array
herr = H5Dclose(data_id);
if(herr<0) cgp_doError;

// Close the group
herr = H5Gclose(group_id);
if(herr<0) cgp_doError;

return 0;
}

// The mallocs and memory copies in here need to be verified
int queue_slice_write(SliceType_t type, int F, int B, int Z, void* SN, int rank,
int* min, int* max, void* data) {
int k;
slice_t* slices = malloc((write_queue_len+1) * sizeof(slice_t));
for(k=0;k<write_queue_len;k++) slices[k] = write_queue[k];
slice_t* slice = &slices[write_queue_len];
slice->type = type;
slice->F = F;
slice->B = B;
slice->Z = Z;
slice->rank = rank;
slice->min = (int*) malloc(rank*sizeof(int));
for(k=0;k<rank;k++) slice->min[k] = min[k];
slice->max = (int*) malloc(rank*sizeof(int));
for(k=0;k<rank;k++) slice->max[k] = max[k];
slice->data = data;
if(type!=Array&&type!=Empty) slice->Selector = *((int*) SN);
else if(type==Array) strcpy(slice->name,(char*) SN);
else {}
if(write_queue!=NULL) free(write_queue);
write_queue = slices;
write_queue_len++;
return 0;

// The mallocs and memory copies in here need to be verified
int queue_flush(void) {
  int err;
herr_t herr;
  int i, j, k;
  int max_queue_len = 0;
  int world_rank = 0;
  int world_size = 0;
  err = MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  err = MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  err = MPI_Allreduce(&write_queue_len, &max_queue_len, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
if (max_queue_len==0) max_queue_len = 1;

slice_t queue[max_queue_len];

for(k=0;k<write_queue_len;k++) {
    queue[k] = write_queue[k];
    if(queue[k].type==Array) strcpy(queue[k].name,write_queue[k].name);
}

for(k=write_queue_len;k<max_queue_len;k++) {
    queue[k].type = Empty;
    queue[k].F = -1;
}

int fn = queue[0].F;
int Fs[world_size];

err = MPI_Allgather(&fn, 1,MPI_INT, Fs, world_size, MPI_INT, MPI_COMM_WORLD);

for(k=0;k<max_queue_len;k++) {
    hid_t data_id;
    switch(queue[k].type) {
    case (Empty):
        data_id = H5Dopen2(files[fn].file_id, " dummy", H5P_DEFAULT);
        hid_t plist_id = H5Pcreate(H5P_DATASET_XFER);
        herr = H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);
        hsize_t dim = 1;
        hid_t shape_id = H5Screate_simple(1,&dim, NULL);
        int buf = 0;
        H5Dwrite(data_id, H5T_NATIVE_INT, shape_id, shape_id, plist_id, &buf);
        H5Sclose(shape_id);
        H5Pclose(plist_id);
        H5Dclose(data_id);
        break;
    case (Coords):
        err = cgp_coord_write_data(queue[k].F,queue[k].B,queue[k].Z,queue[k].Selector,queue[k].min,queue[k].max,queue[k].data);
break;
case(Elements):
    err = cgp_section_write_data(queue[k].F, queue[k].B, queue[k].Z, queue[k].Selector, queue[k].min[0], queue[k].max[0], queue[k].data);
    break;
case(Solution):
    err = cgp_sol_write_data(queue[k].F, queue[k].B, queue[k].Z, queue[k].Selector, queue[k].min, queue[k].max, queue[k].data);
    break;
case(Array):
    err = cgp_array_write_data(queue[k].F, queue[k].B, queue[k].Z, queue[k].name, queue[k].min, queue[k].max, queue[k].data);
    break;
}
for(k=0;k<write_queue_len;k++) {
    if(write_queue[k].min!=NULL) free(write_queue[k].min);
    if(write_queue[k].max!=NULL) free(write_queue[k].max);
}
if(write_queue!=NULL) free(write_queue);
write_queue = NULL;
write_queue_len = 0;
return 0;
}

A.4.11 pcgslib.h File Reference
#include "mpi.h"
#include "hdf5.h"

Include dependency graph for pcgslib.h:

pcgslib.h

mpi.h   hdf5.h
This graph shows which files directly or indirectly include this file:

Defines

- `#define cgp_doError ;`
- `#define printTime`

Typedefs

- `typedef int DataType_t`

Enumerations

- `enum ZoneType_t { Structured, Unstructured }`
- `enum ElementType_t {
  ElementTypeNull, ElementTypeUserDefined, NODE, BAR_2,
  BAR_3, TRI_3, TRI_6, QUAD_4,
  QUAD_8, QUAD_9, TETRA_4, TETRA_10,
  PYRA_5, PYRA_13, PYRA_14, PENTA_6,
  PENTA_15, PENTA_18, HEXA_8, HEXA_20,
  HEXA_27, MIXED, NGON_n, NFACE_n }
- `enum GridLocation_t { Vertex, CellCenter }`
- `enum SliceType_t {
  Empty, Coords, Elements, Solution,
  Array }`
Functions

- int cgp_open (const char *filename, int mode, MPI_Comm comm, MPI_Info *info, int *fn)
- int cgp_close (int fn)
- int cgp_base_read (int fn, int B, char *basename, int *cell_dim, int *phys_dim)
- int cgp_base_write (int fn, char const *basename, int cell_dim, int phys_dim, int *B)
- int cgp_nbases (int fn, int *nbases)
- int cgp_zone_read (int fn, int B, int Z, char *zonename, int *nijk)
- int cgp_zone_type (int fn, int B, int Z, ZoneType_t *zonetype)
- int cgp_zone_write (int fn, int B, const char *zonename, const int *nijk, ZoneType_t type, int *Z)
- int cgp_nzones (int fn, int B, int *nzones)
- int cgp_coord_write (int fn, int B, int Z, DataType_t type, const char *coordname, int *C)
- int cgp_coord_write_data (int fn, int B, int Z, int C, int *min, int *max, void *coord_array)
- int cgp_sol_write (int fn, int B, int Z, char *solname, GridLocation_t location, int *S)
- int cgp_sol_write_data (int fn, int B, int Z, int S, int *min, int *max, void *data)
- int cgp_nsols (int fn, int B, int Z, int *nsols)
- int cgp_section_write (int fn, int B, int Z, char *sectionname, ElementType_t type, int start, int end, int nbndry, int *S)
- int cgp_section_write_data (int fn, int B, int Z, int S, int min, int max, int *elements)
- int cgp_array_write (int fn, int B, int Z, char *arrayname, GridLocation_t location)
- int cgp_array_write_data (int fn, int B, int Z, char *arrayname, int *min, int *max, void *data)
- int queue_slice_write (SliceType_t type, int F, int B, int Z, void *SN, int rank, int *min, int *max, void *data)
- int queue_flush (void)

Variables

- int preallocate
A.4.11.1 Detailed Description

Author:

Kyle Horne <horne.kyle@gmail.com>

Version:

0.2

A.4.11.2 LICENSE

BSD style license

A.4.11.3 DESCRIPTION

Header file for all public functions of the pcgns library

Definition in file pcgnslib.h.

A.4.11.4 Define Documentation

A.4.11.4.1 #define cgp_doError ;

Definition at line 17 of file pcgnslib.h.

A.4.11.4.2 #define printTime

Definition at line 21 of file pcgnslib.h.

A.4.11.5 Typedef Documentation

A.4.11.5.1 typedef int DataType_t

Definition at line 44 of file pcgnslib.h.
A.4.11.6 Enumeration Type Documentation

A.4.11.6.1 enum ElementType_t

Enumerator:

   ElementTypeNull

   ElementTypeUserDefined

   NODE
   BAR_2
   BAR_3
   TRI_3
   TRI_6
   QUAD_4
   QUAD_8
   QUAD_9
   TETRA_4
   TETRA_10
   PYRA_5
   PYRA_13
   PYRA_14
   PENTA_6
   PENTA_15
   PENTA_18
   HEXA_8
   HEXA_20
HEXA_27
MIXED
NGON_n
NFACE_n

Definition at line 31 of file pcgnslib.h.

A.4.11.6.2 enum GridLocation_t

Enumerator:

Vertex
CellCenter

Definition at line 45 of file pcgnslib.h.

A.4.11.6.3 enum SliceType_t

Enumerator:

Empty
Coords
Elements
Solution
Array

Definition at line 46 of file pcgnslib.h.

A.4.11.6.4 enum ZoneType_t
Enumerator:

*Structured*

*Unstructured*

Definition at line 30 of file pcgnslib.h.

A.4.11.7 Function Documentation

A.4.11.7.1 `int cgp_array_write (int fn, int B, int Z, char *arrayname, GridLocation_t location)`

Write an array to a zone

Parameters:

`fn` [in]: Handle of the file

`B` [in]: Index of the base

`Z` [in]: Index of the zone

`arrayname` [in]: Name of array

`location` [in]: Location of solution within each cell

Returns:

Error code

Definition at line 1191 of file pcgnslib.c.

Here is the call graph for this function:

![Call Graph Diagram]
Here is the caller graph for this function:

A.4.11.7.2 int cgp_array_write_data (int \textit{fn}, int \textit{B}, int \textit{Z}, char * \textit{arrayname}, int * \textit{min}, int * \textit{max}, void * \textit{data})

Write an array's data in parallel

Parameters:

\textit{fn} [in]: Handle of the file
\textit{B} [in]: Index of the base
\textit{Z} [in]: Index of the zone
\textit{arrayname} [in]: Name of array
\textit{min} [in]: Lower bound array for data
\textit{max} [in]: Upper bound array for data
\textit{data} [in]: Data to be written

Returns:

Error code

Definition at line 1251 of file pcgnslib.c.

Here is the caller graph for this function:

A.4.11.7.3 int cgp_base_read (int \textit{fn}, int \textit{B}, char * \textit{basename}, int * \textit{cell_dim}, int * \textit{phys_dim})

Read info about a base
Parameters:

\[ fn \] [in]: Handle of the file

\[ B \] [in]: Index of the base

\[ basename \] [out]: Name of the base

\[ cell\_dim \] [out]: Cell dimensions of the base

\[ phys\_dim \] [out]: Physical dimensions of the base

Returns:

Error code

Definition at line 169 of file pcgslib.c.

Here is the call graph for this function:

```
cgp_base_read
 \-----\ node_idx2name
      \--\ node_name_finder
          \--\ get_str_attb
                \--\ num_nodes
                    \--\ node_counter
                        \--\ main
```

Here is the caller graph for this function:

```
cgp_base_read
 \-----\ main
```

A.4.11.7.4 \hspace{1em} int cgp_base_write (int \( fn \), char const * \( basename \), int \( cell\_dim \), int \( phys\_dim \),

\hspace{1em} int * \( B \))

Write a base to a file

Parameters:

\[ fn \] int[in]: Handle of the file

\[ basename \] [in]: Name of the base to write

\[ cell\_dim \] [in]: Cell dimensions of the base

\[ phys\_dim \] [in]: Physical dimensions of the base
**B** [out]: Index of the base

**Returns:**

Error code

Definition at line 234 of file pcgnslib.c.

Here is the call graph for this function:

```
cgp_base_write
  \_ del_node
  \_ free_base
  \_ new_node
  \_ node_exists
  \_ free_zone
  \_ free_int_attb
  \_ new_str_attb
  \_ free_coord
  \_ free_section
  \_ free_sol
```

Here is the caller graph for this function:

```
cgp_base_write
  \_ main
```

A.4.11.7.5  **int cgp_close (int fn)**

Close a previously opened file

**Parameters:**

*fn* [in]: Handle of the file to close

**Returns:**

Error code

Definition at line 159 of file pcgnslib.c.

Here is the call graph for this function:

```
cgp_close
  \_ del_node
  \_ free_file
  \_ free_base
  \_ free_zone
  \_ free_int_attb
  \_ free_coord
  \_ free_section
  \_ free_sol
```
Here is the caller graph for this function:

A.4.11.7.6 int cgp_coord_write (int fn, int B, int Z, DataType_t type, const char * coordname, int * C)

Write coords group, but not data, to a grid

Parameters:

fn [in]: Handle of the file

B [in]: Index of the base

Z [in]: Index of the zone

datatype [in]: Type of floats stored

coordname [in]: Name of the coords

C [out]: Index of the coords

Definition at line 667 of file pcgnslib.c.

Here is the call graph for this function:

A.4.11.7.7 int cgp_coord_write_data (int fn, int B, int Z, int C, int * min, int * max, void * coord_array)

Write coods to a grid in parallel
Parameters:

\( fn \) [in]: Handle of the file

\( B \) [in]: Index of the base

\( Z \) [in]: Index of the zone

\( C \) [in]: Index of the coords

\( \text{range}_\text{min} \) [in]: Array of lower bound index

\( \text{range}_\text{max} \) [in]: Array of upper bound index

\( \text{coord}_\text{array} \) [in]: Pointer to the data

Returns:

Error code

Definition at line 750 of file pcgnslib.c.

Here is the caller graph for this function:

![Caller Graph](image)

A.4.11.7.8  \textbf{int cgp_nbases (int} \textit{fn, int * nbases)}

Read the number of bases in a file

Parameters:

\( fn \) [in]: Handle of the file

\( \text{nbases} \) [out]: Number of bases in the specified file

Returns:

Error code

Definition at line 318 of file pcgnslib.c.
A.4.11.7.9  int cgp_nsols (int fn, int B, int Z, int * nsols)

Read the number of solutions in a zone

Parameters:

fn [in]: Handle of the file
B [in]: Index of the base
Z [in]: Index of the zone

nsols [out]: Number of solutions in the specified zone

Returns:

Error code

Definition at line 960 of file pcgnslib.c.

A.4.11.7.10 int cgp_nzones (int fn, int B, int * nzones)

Read the number of zones in a base

Parameters:

fn [in]: Handle of file
B [in]: Index of base

nzones [out]: Number of zones in the specified base

Returns:

Error code

Definition at line 660 of file pcgnslib.c.

A.4.11.7.11 int cgp_open (const char * filename, int mode, MPI_Comm comm, MPI_Info * info, int * fn)

Open a file for reading and writing
Parameters:

*filename* [in]: Name of the file to open

*mode* [in]: IO mode (read/write)

*comm* [in]: MPI communicator on which to open the file

*info* [in]: MPI info object to allow hints passed to MPI-IO

*fn* [out]: Handle of the opened file

Returns:

Error code

Definition at line 42 of file pcgnslib.c.

Here is the call graph for this function:

![Call graph for cgp_open function](image)

Here is the caller graph for this function:

![Caller graph for cgp_open function](image)
A.4.11.7.12 int cgpssection_write (int fn, int B, int Z, char * sectionname, ElementType_t type, int start, int end, int nbndry, int * S)

Write the element connectivity groups for a section

Parameters:

fn [in]: Handle of the file

B [in]: Index of the base

Z [in]: Index of the zone

C [in]: Index of the coords

sectionname [in]: Name of element section

type [in]: Type of element data

start [in]: Element lower bound index

end [in]: Element upper bound index

nbndry [in]: Number of boundary elements (unused)

S [out]: Section index

Returns:

Error code

Definition at line 976 of file pcgnslib.c.

Here is the call graph for this function:

Here is the caller graph for this function:
A.4.11.7.13  int cgp_section_write_data (int \textit{fn}, \textit{int B}, \textit{int Z}, \textit{int S}, \textit{int min}, \textit{int max}, \textit{int \ast elements})

Write the element connectivity data for a section

Parameters:

\textit{fn} [in]: Handle of the file

\textit{B} [in]: Index of the base

\textit{Z} [in]: Index of the zone

\textit{C} [in]: Index of the coords

\textit{S} [in]: Section index

\textit{min} [in]: Output array lower bound index

\textit{max} [in]: Output array upper bound index

\textit{elements} [in]: Pointer to the data

Returns:

Error code

Definition at line 1126 of file pcgnslib.c.

Here is the caller graph for this function:

A.4.11.7.14  int cgp_sol_write (int \textit{fn}, \textit{int B}, \textit{int Z}, \textit{char \ast solname}, \textit{GridLocation_t location}, \textit{int \ast S})

Write a solution to a zone

Parameters:

\textit{fn} [in]: Handle of the file
B [in]: Index of the base

Z [in]: Index of the zone

solname [in]: Name of solution

location [in]: Location of solution within each cell

S [out]: Index of solution

Returns:

Error code

Definition at line 876 of file pcnslib.c.

Here is the call graph for this function:

Here is the caller graph for this function:

A.4.11.7.15 int cgp_sol_write_data (int fn, int B, int Z, int S, int * min, int * max, void * data)

Parameters:

fn [in]: Handle of the file

B [in]: Index of the base

Z [in]: Index of the zone

S [in]: Index of solution
\textbf{min} [in]: Lower bound array for data

\textbf{max} [in]: Upper bound array for data

\textbf{data} [in]: Data to be written

\textbf{Returns:}

Error code

Definition at line 813 of file pcgnslib.c.

Here is the caller graph for this function:

\begin{center}
\includegraphics[width=0.5\textwidth]{graph.png}
\end{center}

\textbf{A.4.11.7.16} \hspace{1em} \texttt{int cg\_zone\_read (int \textit{fn}, int \textit{B}, int \textit{Z}, char * \textit{zonename}, int * \textit{nijk})}

Read info about a zone

\textbf{Parameters:}

\textit{fn} [in]: Handle of the file

\textit{B} [in]: Index of the base

\textit{Z} [in]: Index of the zone to read

\textit{zonename} [out]: Name of the zone

\textit{nijk} [out]: Dimensions of the zone

\textbf{Returns:}

Error code

Definition at line 325 of file pcgnslib.c.
Here is the call graph for this function:

```
\begin{tikzpicture}
  \node (main) {main};
  \node (zone_read) [above left of=main] {cgp_zone_read};
  \draw[->] (main) -- (zone_read);
\end{tikzpicture}
```

Here is the caller graph for this function:

```
\begin{tikzpicture}
  \node (zone_read) {cgp_zone_read};
  \node (main) [above right of=zone_read] {main};
  \draw[->] (zone_read) -- (main);
\end{tikzpicture}
```

### A.4.11.7.17 int cgp_zone_type (int fn, int B, int Z, ZoneType_t * zonetype)  

Read the type of a zone

**Parameters:**

- **fn** [in]: Handle of the file
- **B** [in]: Index of the base
- **Z** [in]: Index of the zone
- **zonetype** [out]: Type of zone

**Returns:**

Error code

Definition at line 535 of file pcgnslib.c.

### A.4.11.7.18 int cgp_zone_write (int fn, int B, const char * zonename, const int * nijk, ZoneType_t type, int * Z)  

Write a zone to a base

**Parameters:**

- **fn** [in]: Handle of the file
**B** [in]: Index of the base

**zonename** [in]: Name of the zone to write

**nijk** [in]: Dimensions of the zone

**type** [in]: Type of zone

**Z** [out]: Index of the zone

**Returns:**

Error code

Definition at line 544 of file pcgnslib.c.

Here is the call graph for this function:

Here is the caller graph for this function:

A.4.11.7.19  **int queue_flush (void)**

Flush all the IO operations waiting in the queue

**Returns:**

Error code

Definition at line 1344 of file pcgnslib.c.
Here is the call graph for this function:

![Call Graph](call_graph.png)

Here is the caller graph for this function:

![Caller Graph](caller_graph.png)

A.4.11.7.20 int queue_slice_write (SliceType_t type, int F, int B, int Z, void *SN, int rank, int *min, int *max, void *data)

Queue an IO write operation for flushing later

Parameters:

- **type** [in]: Type of operation to queue
- **F** [in]: Handle of the file
- **B** [in]: Index of the base
- **Z** [in]: Index of the zone
- **SN** [in]: Pointer to array locator, which is an int for coordinates, solutions and sections, but a string for arrays
- **rank** [in]: Rank of data to be written
- **min** [in]: Pointer to the minimum location array
- **max** [in]: Pointer to the maximum location array
- **data** [in]: Pointer to the data to be written

Returns:

Error code
Definition at line 1315 of file pcgnslib.c.
Here is the caller graph for this function:

queue_slice_write → main

A.4.11.8 Variable Documentation

A.4.11.8.1 int preallocate

Definition at line 35 of file pcgnslib.c.

A.4.12 pcgnslib.h

```c
#ifndef PCGNSLIB_H_
#define PCGNSLIB_H_
#endif _DEBUG

#define cgp_doError {printf("Error at %s:%u\n", __FILE__, __LINE__); return 1;}
#define printTime printf("Time at %s:%u = %f\n", __FILE__, __LINE__, MPI_Wtime())
```
```
#define printTime

#include "mpi.h"
#include "hdf5.h"

//=====================//
//== Begin Datatypes ==//
//=====================//

typedef enum {Structured, Unstructured} ZoneType_t;
typedef enum {
    ElementTypeNull, ElementTypeUserDefined, // 0 1
    NODE, // 2
    BAR_2, BAR_3, // 3 4
    TRI_3, TRI_6, // 5 6
    QUAD_4, QUAD_8, QUAD_9, // 7 8 9
    TETRA_4, TETRA_10, // 10 11
    PYRA_5, PYRA_13, PYRA_14, // 12 13 14
    PENTA_6, PENTA_15, PENTA_18, // 15 16 17
    HEXA_8, HEXA_20, HEXA_27, // 18 19 20
    MIXED, // 21
    NGON_n, NFACE_n // 22 23
} ElementType_t;
typedef int DataType_t;
typedef enum {Vertex, CellCenter} GridLocation_t;
typedef enum {Empty, Coords, Elements, Solution, Array} SliceType_t;

extern int preallocate;

//===============================//
//== Begin Function Prototypes ==//
//===============================//

 //= File IO Prototypes =//

int cgp_open(const char* filename, int mode, MPI_Comm comm, MPI_Info info, int*
int cgp_close(int fn);

int cgp_base_read(int fn, int B, char* basename, int* cell_dim, int* phys_dim);
int cgp_base_write(int fn, char const* basename, int cell_dim, int phys_dim, int* B);
int cgp_nbases(int fn, int *nbases);

int cgp_zone_read(int fn, int B, int Z, char* zonename, int* nijk);
int cgp_zone_type(int fn, int B, int Z, ZoneType_t *zonetype);
int cgp_zone_write(int fn, int B, const char* zonename, const int* nijk, ZoneType_t type, int* Z);
int cgp_nzones(int fn, int B, int *nzones);

int cgp_coord_write(int fn, int B, int Z, DataType_t type, const char* coordname, int* C);
int cgp_coord_write_data(int fn, int B, int Z, int C, int* min, int* max, void* coord_array);

int cgp_sol_write(int fn, int B, int Z, char* solname, GridLocation_t location, int *S);
int cgp_sol_write_data(int fn, int B, int Z, int S, int* min, int* max, void* dat
int cgp_nsols(int fn, int B, int Z, int* nsols);

//= Unstructured Grid Prototypes =//
int cgp_section_write(int fn, int B, int Z, char* sectionname, ElementType_t type ,
int start, int end, int nbndry, int* S);

int cgp_section_write_data(int fn, int B, int Z, int S, int min, int max, int *elements);

//= Array IO Prototypes =//
int cgp_array_write(int fn, int B, int Z, char *arrayname, GridLocation_t location);

int cgp_array_write_data(int fn, int B, int Z, char *arrayname, int* min, int* max, void* data);

//= Queue IO Prototypes =//
int queue_slice_write(SliceType_t type, int F, int B, int Z, void* SN, int rank,
int* min, int* max, void* data);

int queue_flush(void);

A.4.13 test_base.c File Reference

#include "pcgnslib.h"
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"
Include dependency graph for test_base.c:

```
functions

- int main (int argc, char *argv[])
```

A.4.13.1 Detailed Description

Author:

Kyle Horne <horne.kyle@gmail.com>

Version:

0.2

A.4.13.2 LICENSE

BSD style license

A.4.13.3 DESCRIPTION

Test program for pcgns library

Definition in file test_base.c.

A.4.13.4 Function Documentation

A.4.13.4.1 int main (int argc, char *argv[])

Definition at line 17 of file test_base.c.

Here is the call graph for this function:

A.4.14 test_base.c

00001
00002
00003
00004
00005
00006
00007
00008
00009
00010
00011 #include "pcgnslib.h"
00012
```c
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"

int main(int argc, char* argv[]) {
    int err;
    int comm_size;
    int comm_rank;
    MPI_Info info;
    int fn;
    int B;
    char basename[100];
    int cell_dim = 3;
    int phys_dim = 3;

    err = MPI_Init(&argc,&argv);
    if(err!=MPI_SUCCESS) cgp_doError;
    err = MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    if(err!=MPI_SUCCESS) cgp_doError;
    err = MPI_Comm_rank(MPI_COMM_WORLD, &comm_rank);
    if(err!=MPI_SUCCESS) cgp_doError;
    err = MPI_Info_create(&(info));
    if(err!=MPI_SUCCESS) cgp_doError;
    err = cgp_open("test_base.cgns", 0, MPI_COMM_WORLD, &info, &fn);
    if(err!=0) cgp_doError;
    err = cgp_base_write(fn, "Base 1", cell_dim, phys_dim, &B);
    if(err!=0) cgp_doError;
    err = cgp_base_read(fn, B, basename, &cell_dim, &phys_dim);
    if(err!=0) cgp_doError;
    err = cgp_close(fn);
    if(err!=0) cgp_doError;
    err = MPI_Finalize();
    if(err!=MPI_SUCCESS) cgp_doError;
}
```
A.4.15 test_queue.c File Reference

```c
#include "pcgnslib.h"
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"
```

Include dependency graph for test_queue.c:

```
Functions

- int main (int argc, char ∗argv[ ])

A.4.15.1 Detailed Description

Author:

Kyle Horne <horne.kyle@gmail.com>

Version:

0.2

A.4.15.2 LICENSE

BSD style license

A.4.15.3 DESCRIPTION
Test program for pcgns library

Definition in file test_queue.c.

A.4.15.4 Function Documentation

A.4.15.4.1 int main (int argc, char *argv[ ])

Definition at line 17 of file test_queue.c.

Here is the call graph for this function:

![Call Graph for int main()](image-url)
A.4.16 test_queue.c

#include "pcgnslib.h"
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"

int main(int argc, char* argv[]) {
    int err;
    int comm_size;
    int comm_rank;
    MPI_Info info;
    int fn;
    int B;
    int 2;
    char basename[100];
    char zonename[100];
    int cell_dim = 3;
    int phys_dim = 3;
    int nijk[3][3];

    nijk[0][0] = 10;
    nijk[0][1] = 10;
    nijk[0][2] = 1;
    nijk[1][0] = nijk[0][0]-1;
nijk[1][1] = nijk[0][1]-1;
nijk[1][2] = nijk[0][2]-1;
nijk[2][0] = 0;
nijk[2][1] = 0;
nijk[2][2] = 0;

er = MPI_Init(&argc,&argv);
if(err!=MPI_SUCCESS) cgp_doError;

err = MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
if(err!=MPI_SUCCESS) cgp_doError;

err = MPI_Comm_rank(MPI_COMM_WORLD, &comm_rank);
if(err!=MPI_SUCCESS) cgp_doError;

err = MPI_Info_create(&(info));
if(err!=MPI_SUCCESS) cgp_doError;

err = cgp_open("test_queue.cgns", 0, MPI_COMM_WORLD, &info, &fn);
if(err!=0) cgp_doError;

err = cgp_base_write(fn, "Base 1", cell_dim, phys_dim, &B);
if(err!=0) cgp_doError;

err = cgp_zone_write(fn, B, "Zone 1", &(nijk[0][0]), Structured, &Z);
if(err!=0) cgp_doError;

err = cgp_zone_read(fn, B, Z, zonename, &(nijk[0][0]));
if(err!=0) cgp_doError;

int min = 0;
int max = 0;
double data = 0.0;
err = queue_slice_write(Empty, fn, B, Z, NULL, 1, &min, &max, &data);
err = queue_flush();

err = cgp_close(fn);
if(err!=0) cgp_doError;

err = MPI_Finalize();
if(err!=MPI_SUCCESS) cgp_doError;
A.4.17  test_unstructured.c File Reference

```c
#include "pcgnslib.h"
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"
```

Include dependency graph for test_unstructured.c:

Functions

- int main (int argc, char ∗argv[ ])

A.4.17.1  Detailed Description

Author:

Kyle Horne <horne.kyle@gmail.com>

Version:

0.2

A.4.17.2  LICENSE

BSD style license

A.4.17.3  DESCRIPTION
Test program for pcgns library
Definition in file test_unstructured.c.

### A.4.17.4 Function Documentation

#### A.4.17.4.1 int main (int argc, char ∗ argv[])
A.4.18  test_unstructured.c

```c
#include "pcgnslib.h"
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"

int main(int argc, char* argv[]) {
    int err;
    int comm_size;
    int comm_rank;
    MPI_Info info;
    int fn;
    int B;
    int Z;
    int S;
    char basename[100];
    char zonename[100];
    int cell_dim = 3;
    int phys_dim = 3;
    int nijk[3][1];
    int k;
    nijk[0][0] = 10;
    nijk[1][0] = nijk[0][0]-1;
```
nijk[2][0] = 0;

er = MPI_Init(&argc,&argv);
if(err!=MPI_SUCCESS) cgp_doError;
er = MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
if(err!=MPI_SUCCESS) cgp_doError;
er = MPI_Comm_rank(MPI_COMM_WORLD, &comm_rank);
if(err!=MPI_SUCCESS) cgp_doError;
er = MPI_Info_create(&(info));
if(err!=MPI_SUCCESS) cgp_doError;
er = cgp_open("test_unstructured.cgns", 0, MPI_COMM_WORLD, &info, &fn);
if(err!=0) cgp_doError;
er = cgp_base_write(fn, "Base 1", cell_dim, phys_dim, &B);
if(err!=0) cgp_doError;
er = cgp_zone_write(fn, B, "Zone 1", &(nijk[0][0]), Unstructured, &Z);
if(err!=0) cgp_doError;

double x[10/comm_size];
double y[10/comm_size];
double z[10/comm_size];

int min[1] = {10/comm_size*comm_rank};
int max[1] = {10/comm_size*(comm_rank+1)-1};

for(k=0;k<10/comm_size;k++) {
    x[k] = (double) (min[0]+k);
    y[k] = 0.0;
    z[k] = 0.0;
}

int Cx,Cy,Cz;

er = cgp_coord_write(fn,B,Z,Unstructured,"CoordinateX",&Cx);
err = cgp_coord_write_data(fn,B,Z,Cz,min,max,x);
err = cgp_coord_write(fn, B, Z, Unstructured, "CoordinateY", &Cy);

//~ err = cgp_coord_write_data(fn, B, Z, Cy, min, max, y);

err = cgp_coord_write(fn, B, Z, Unstructured, "CoordinateZ", &Cz);

//~ err = cgp_coord_write_data(fn, B, Z, Cz, min, max, z);

err = queue_slice_write(Coords, fn, B, Z, &Cx, 1, min, max, x);
err = queue_slice_write(Coords, fn, B, Z, &Cy, 1, min, max, y);
err = queue_slice_write(Coords, fn, B, Z, &Cz, 1, min, max, z);
err = queue_flush();

int start = 1;
int end = 9;

int nelems = (comm_rank != comm_size-1)?9/comm_size:9-(9/comm_size)*(comm_size-1);

printf("%d:%d
", comm_rank, nelems);

int emin = (9/comm_size)*comm_rank+1;
int emax = (comm_rank != comm_size-1)?(9/comm_size)*(comm_rank+1):9;

int elements[nelems*2];

for(k=0;k<nelems;k++) {
    elements[2*k] = k+emin;
    elements[2*k+1] = k+emin+1;
}

printf("%d:%d %d %d
", comm_rank, nelems, emin, emax);

//~ err = cgp_section_write_data(fn, B, Z, S, emin, emax, &elements[0]);

err = queue_slice_write(Elements, fn, B, Z, &S, 1, &emin, &emax, elements);
err = queue_flush();
err = cgp_close(fn);

if(err!=0) cgp_doError;
A.4.19 test_zone.c File Reference

```c
#include "pcgnslib.h"
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"
#include "hdf5.h"
```

Include dependency graph for test_zone.c:

A.4.19.2 LICENSE

BSD style license
A.4.19.3 DESCRIPTION

Test program for pcgns library
Definition in file test_zone.c.

A.4.19.4 Function Documentation

A.4.19.4.1 int main (int argc, char *argv[])

Definition at line 17 of file test_zone.c.
Here is the call graph for this function:
```c
#include "pcgnslib.h"
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"

int main(int argc, char* argv[]) {
    int err;
    int comm_size;
    int comm_rank;
    MPI_Info info;
    int fn;
    int B;
    int Z;
    char basename[100];
    char zonename[100];
    int cell_dim = 3;
    int phys_dim = 3;
    int nijk[3][3];

    nijk[0][0] = 10;
    nijk[0][1] = 10;
    nijk[0][2] = 1;
    nijk[1][0] = nijk[0][0]-1;
    nijk[1][1] = nijk[0][1]-1;
```
nijk[1][2] = nijk[0][2]-1;
nijk[2][0] = 0;
nijk[2][1] = 0;
nijk[2][2] = 0;

err = MPI_Init(&argc,&argv);
if(err!=MPI_SUCCESS) cgp_doError;
err = MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
if(err!=MPI_SUCCESS) cgp_doError;
err = MPI_Comm_rank(MPI_COMM_WORLD, &comm_rank);
if(err!=MPI_SUCCESS) cgp_doError;
err = MPI_Info_create(&(info));
if(err!=MPI_SUCCESS) cgp_doError;
err = cgp_open("test_zone.cgns", 0, MPI_COMM_WORLD, &info, &fn);
if(err!=0) cgp_doError;
err = cgp_base_write(fn, "Base 1", cell_dim, phys_dim, &B);
if(err!=0) cgp_doError;
err = cgp_zone_write(fn, B, "Zone 1", &(nijk[0][0]), Structured, &Z);
if(err!=0) cgp_doError;
err = cgp_zone_read(fn, B, Z, zonename, &(nijk[0][0]));
if(err!=0) cgp_doError;
err = cgp_close(fn);
if(err!=0) cgp_doError;
err = MPI_Finalize();
if(err!=MPI_SUCCESS) cgp_doError;
return 0;

A.4.21 thesis_benchmark.c File Reference

#include "pcgnslib.h"
#include "stdio.h"
#include "stdlib.h"
```c
#include "math.h"
#include "mpi.h"

Include dependency graph for thesis_benchmark.c:

Functions

- int read_inputs (int *argc, char ***argv)
- int initialize (int *argc, char ***argv)
- int finalize (void)
- int main (int argc, char *argv[ ])

Variables

- int comm_size
- int comm_rank
- MPI_Info info
- double data_size
- int N
- int Nl
- int pc
- int zpp
- int ppz
- int zc
- int * zones
```
• int * subzones
• double * x
• double * y
• double * z
• int * e
• double * u
• double * v
• double * w
• double * h

A.4.21.1 Function Documentation

A.4.21.1.1 int finalize (void)

Definition at line 119 of file thesis_benchmark.c.

A.4.21.1.2 int initialize (int * argc, char *** argv)

Definition at line 62 of file thesis_benchmark.c.

Here is the call graph for this function:

A.4.21.1.3 int main (int argc, char * argv[])

Definition at line 135 of file thesis_benchmark.c.
Here is the call graph for this function:

A.4.21.1.4 int read_inputs (int * argc, char *** argv)

Definition at line 34 of file thesis_benchmark.c.

Here is the caller graph for this function:

A.4.21.2 Variable Documentation
A.4.21.2.1  int comm_rank

Definition at line 8 of file thesis_benchmark.c.

A.4.21.2.2  int comm_size

Definition at line 7 of file thesis_benchmark.c.

A.4.21.2.3  double data_size

Definition at line 11 of file thesis_benchmark.c.

A.4.21.2.4  int* e

Definition at line 26 of file thesis_benchmark.c.

A.4.21.2.5  double* h

Definition at line 32 of file thesis_benchmark.c.

A.4.21.2.6  MPI_Info info

Definition at line 9 of file thesis_benchmark.c.

A.4.21.2.7  int N

Definition at line 12 of file thesis_benchmark.c.
A.4.21.2.8 int Nl

Definition at line 13 of file thesis_benchmark.c.

A.4.21.2.9 int pc

Definition at line 14 of file thesis_benchmark.c.

A.4.21.2.10 int ppz

Definition at line 16 of file thesis_benchmark.c.

A.4.21.2.11 int* subzones

Definition at line 20 of file thesis_benchmark.c.

A.4.21.2.12 double* u

Definition at line 28 of file thesis_benchmark.c.

A.4.21.2.13 double* v

Definition at line 29 of file thesis_benchmark.c.

A.4.21.2.14 double* w

Definition at line 30 of file thesis_benchmark.c.
A.4.21.2.15 double* x

Definition at line 22 of file thesis_benchmark.c.

A.4.21.2.16 double* y

Definition at line 23 of file thesis_benchmark.c.

A.4.21.2.17 double* z

Definition at line 24 of file thesis_benchmark.c.

A.4.21.2.18 int zc

Definition at line 17 of file thesis_benchmark.c.

A.4.21.2.19 int* zones

Definition at line 19 of file thesis_benchmark.c.

A.4.21.2.20 int zpp

Definition at line 15 of file thesis_benchmark.c.

A.4.22 thesis_benchmark.c

00001 #include "pcgnslib.h"
00002 #include "stdio.h"
00003 #include "stdlib.h"
```c
#include "math.h"
#include "mpi.h"

int comm_size;
int comm_rank;
MPI_Info info;

double data_size;
int N;
int Nl;
int pc;
int zpp;
int ppz;
int zc;

int* zones;
int* subzones;

double* x;
double* y;
double* z;

double* e;

double* u;
double* v;
double* w;

double* h;

int read_inputs(int* argc, char*** argv) {
    int k;
    if (comm_rank==0) {
        if (*argc<7) exit(1);
        if (*argv<7) exit(1);
        for (k=1;k<*argc;k++) {
            // Code continues here...
        }
    }
    // Code continues here...
}
```
if(strcmp((*(argv))[k],"-ds")==0) {
    k++;
    sscanf((*(argv))[k],"%lf",&data_size);
    printf("data_size=%lf\n",data_size);
}

if(strcmp((*(argv))[k],"-zpp")==0) {
    k++;
    sscanf((*(argv))[k],"%d",&zpp);
    printf("zpp=%d\n",zpp);
}

if(strcmp((*(argv))[k],"-ppz")==0) {
    k++;
    sscanf((*(argv))[k],"%d",&ppz);
    printf("ppz=%d\n",ppz);
}
}

MPI_Bcast(&data_size,1,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast(&zpp,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(&ppz,1,MPI_INT,0,MPI_COMM_WORLD);
return 0;

int initialize(int* argc, char*** argv) {
    int j,k;
    MPI_Init(argc,argv);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &comm_rank);
    MPI_Info_create(&info);
    read_inputs(argc,argv);
    N = (data_size*1024*1024)/((double) sizeof(double));
    pc = comm_size;
zc = (pc*zpp)/ppz;

Nl = N/comm_size/zpp;

zones = malloc(Nl*sizeof(int));
subzones = malloc(zpp*sizeof(int));

for(k=0;k<zpp;k++) {
    zones[k] = comm_rank/ppz*zpp+k;
    subzones[k] = comm_rank%ppz;
}

// Initialize Arrays
x = malloc(Nl*sizeof(double));
y = malloc(Nl*sizeof(double));
z = malloc(Nl*sizeof(double));
e = malloc(Nl*sizeof(int));

u = malloc(Nl*sizeof(double));
v = malloc(Nl*sizeof(double));
w = malloc(Nl*sizeof(double));
h = malloc(Nl*sizeof(double));

double theta;
double r;

for(k=0;k<Nl;k++) {
    j = Nl*subzones[0]+k;
    theta = ((double) j)/((double) Nl*zpp);
    r = theta;
    x[k] = r*cos(theta);
    y[k] = r*sin(theta);
    z[k] = r;
    e[k] = j+1;
    u[k] = x[k];
    v[k] = y[k];
w[k] = z[k];
h[k] = r;
}

// printf("%d: Nl %d\n", comm_rank, Nl);
// for(k=0;k<zpp;k++) printf("%d: Z%d.%d\n", comm_rank, zones[k], subzones[k]);
return 0;

int finalize(void) {
  free(zones);
  free(subzones);
  free(x);
  free(y);
  free(z);
  free(e);
  free(u);
  free(v);
  free(w);
  MPI_Finalize();
  return 0;
}

int main(int argc, char* argv[]) {
  int k;
  int F;
  int B;
  int nijk[3][1];
  initialize(&argc,&argv);

cgp_open("thesis_benchmark.cgns",0,MPI_COMM_WORLD, &info, &F);

cgp_base_write(F,"Base",3,3,&B);

nijk[0][0] = Nl*ppz;
nijk[1][0] = Nl*ppz;
nijk[2][0] = 0;

int Z[zc];
int Cx[zc];
int Cy[zc];
int Cz[zc];
int E[zc];
int Su[zc];
int Sv[zc];
int Sw[zc];

for(k=0;k<zc;k++) {
    char zonename[100+1];
sprintf(zonename,"%s %d","Zone",k);
cgp_zone_write(F,B,zonename,&(nijk[0][0]),Unstructured,&(Z[k]));
cgp_coord_write(F,B,Z[k],0,"CoordinateX",&(Cx[k]));
cgp_coord_write(F,B,Z[k],0,"CoordinateY",&(Cy[k]));
cgp_coord_write(F,B,Z[k],0,"CoordinateZ",&(Cz[k]));
cgp_section_write(F,B,Z[k],"Elements",NODE,1,Nl*ppz,0,&(E[k]));
cgp_sol_write(F,B,Z[k],"MomentumX",Vertex,&(Su[k]));
cgp_sol_write(F,B,Z[k],"MomentumY",Vertex,&(Sv[k]));
cgp_sol_write(F,B,Z[k],"MomentumZ",Vertex,&(Sw[k]));
cgp_array_write(F,B,Z[k],"phi",Vertex);
}

double T0,T1;
double t0,t1;

MPI_Barrier(MPI_COMM_WORLD);
T0 = MPI_Wtime();
MPI_Barrier(MPI_COMM_WORLD);

t0 = MPI_Wtime();

for(k=0;k<zpp;k++) {
    int min = subzones[k]*Nl;
    int max = (subzones[k]+1)*Nl-1;
    cgp_coord_write_data(F,B,Z[zones[k]],Cx[zones[k]],&min,&max,x);
    cgp_coord_write_data(F,B,Z[zones[k]],Cy[zones[k]],&min,&max,y);
    cgp_coord_write_data(F,B,Z[zones[k]],Cz[zones[k]],&min,&max,z);
}
MPI_Barrier(MPI_COMM_WORLD);

t1 = MPI_Wtime();

if(comm_rank==0) {
    printf("Coords\n");
    printf("tTime=%lf\n",comm_rank,t1-t0);
    printf("tBandwidth=%lf\n",comm_rank,3.0*data_size/(t1-t0));
}

MPI_Barrier(MPI_COMM_WORLD);

t0 = MPI_Wtime();

for(k=0;k<zpp;k++) {
    int min = subzones[k]*Nl;
    int max = (subzones[k]+1)*Nl-1;
    cgp_sol_write_data(F,B,Z[zones[k]],Su[zones[k]],&min,&max,u);
    cgp_sol_write_data(F,B,Z[zones[k]],Sv[zones[k]],&min,&max,v);
    cgp_sol_write_data(F,B,Z[zones[k]],Sw[zones[k]],&min,&max,w);
}
MPI_Barrier(MPI_COMM_WORLD);

t1 = MPI_Wtime();

if(comm_rank==0) {
    printf("Solutions\n");
    printf("tTime=%lf\n",t1-t0);
    printf("tBandwidth=%lf\n",3.0*data_size/(t1-t0));
MPI_Barrier(MPI_COMM_WORLD);
t0 = MPI_Wtime();
for (k=0;k<zpp;k++) {
  int min = subzones[k]*Nl;
  int max = (subzones[k]+1)*Nl-1;
  cgp_array_write_data(F,B,Z[zones[k]],"phi",&min,&max,h);
}
MPI_Barrier(MPI_COMM_WORLD);
t1 = MPI_Wtime();
if(comm_rank==0) {
  printf("Arrays\n");
  printf("tTime=%lf\n",t1-t0);
  printf("tBandwidth=%lf\n",data_size/(t1-t0));
}
MPI_Barrier(MPI_COMM_WORLD);
t0 = MPI_Wtime();
for (k=0;k<zpp;k++) {
  int min = subzones[k]*Nl;
  int max = (subzones[k]+1)*Nl-1;
  min++;
  max++;
  cgp_section_write_data(F,B,Z[zones[k]],E[zones[k]],min,max,e);
}
MPI_Barrier(MPI_COMM_WORLD);
t1 = MPI_Wtime();
if(comm_rank==0) {
  printf("Elements\n");
  printf("tTime=%lf\n",t1-t0);
  printf("tBandwidth=%lf\n",((double) sizeof(int))/((double) sizeof(double)) *data_size/(t1-t0));
MPI_Barrier(MPI_COMM_WORLD);
t0 = MPI_Wtime();
cgp_close(F);
MPI_Barrier(MPI_COMM_WORLD);
t1 = MPI_Wtime();
if(comm_rank==0) printf("Close_Time=%lf\n", t1-t0);

MPI_Barrier(MPI_COMM_WORLD);
T1 = MPI_Wtime();
if(comm_rank==0) {
    printf("Total\n");
    printf("Total Time=%lf\n", T1-T0);
    printf("Total Bandwidth=%lf\n", (6.0+((double) sizeof(int))/((double) sizeof(double)))*data_size/(T1-T0));
}
finalize();
return 0;