A Short Performance Comparison of CFD/CSM Data Management Systems

SMR S.A
2500 Bienne 4
Switzerland
http://www.smr.ch

Version 0.1

SMR Engineering & Development

Copyright © 2001-2002 by SMR S.A. All rights reserved.

NOTICE

The contents of this document may be subject to change at any time and without prior notice. SMR S.A. assumes no responsibility for any errors that may appear in this document and makes no warranty expressly or implied. SMR S.A. shall not be held to any liability with respect to any claim arising from the use of this document or the software described in this document.

Parts of the software described in this document are furnished under a contract and may only be used in accordance with the terms stated in the contract. All rights of this document are reserved. It may not, in whole or part, be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior consent, in writing, from SMR S.A.

A Short Performance Comparison of CFD/CSM Data Management Systems

Version 0.1, September 10, 2002

SMR Engineering & Development
P. O. Box 4014
CH-2500 Bienne 4
Switzerland

http://www.smr.ch

Contents

1 Introduction **1**

2 The Lower Layer: Data Storage and Access **2**

2.1 Functionalities 2

2.2 ADF 3

2.3 NetCDF 3

2.4 MemCom 3

3 Performance Tests of *CGNS*, *MemCom*, and *NetCDF* **5**

3.1 Benchmark Programs 5

3.2 Testing Environment 7

3.3 Create and Write One Dataset 7

3.4 Create and Write Many Datasets 8

3.5 Mixed Benchmark 10

4 The Upper Layer: Data Organisation and API **11**

4.1 Requirements 11

4.2 *CGNS* 12

4.3 The *Julius* Data Structure and Database Definition 12

5 Conclusion **18**

A Short Demonstration of the *MemCom* C API for *jtk* **19**

B A jpmac Parallel Access Example Program **25**

C The Source Code of the MemCom Benchmark Program **30**

D The Source Code of the NetCDF Benchmark Program **33**

E The Source Code of the CGNS Benchmark Program **37**

Chapter 1

Introduction

The B2000 FEM problem solving environment is built on the emphMemCom data manager for the organization and access of computed data involved in multi-disciplinary simulations. In this report an assessment is made of the performance of the *MemCom* data manager and of two public domain data management systems (*CGNS/ADF* and *NetCDF*). Tests haven been selected to reflect typical situations during unsteady computations or computations involving large meshes.

The storage of CFD and CSM data poses two problems (1) the storage of the data itself and (2) the organisation of the different data items in the same database as well as the API which enforces such data organisation. All known solutions separate these tasks in two different layers, where the *lower layer* provides storage of multiple data items, and where the *upper layer* takes care of the organisation of these data items in the database and provides a programming interface tailored to the computation problem (such as CFD). As a result, for each of the three data management systems considered in this report, we will compare the lower and upper layer separately.

In the following Chapters, results of benchmarks for three data management systems are presented. Particular attention is given to the scalability of the systems. The data management systems considered are emphCGNS (with emphADF as lower layer), emphNetCDF, and emphMemCom (with the Julius toolkit (emphjtk) as upper layer). While emphMemCom is proprietary software of SMR, emphCGNS and emphNetCDF are freely available.

Chapter 2

The Lower Layer: Data Storage and Access

2.1 Functionalities

The requirements anf functionalities for the lower layer tests of the three different systems are the following:

- The code for the data storage must be portable. This requirement is met by all three systems.
- The storage format must be platform-independent. This requirement is also met by all three systems.
- Datasets can be attributed with name/value pairs. Only *MemCom* is able to do this.
- Parallel Access via *MPI*. There is a working prototype of a *MPI-MemCom*. *ADF* and *NetCDF* do not provide this.
- Scattered read for data extraction should be as fast as possible. Only *MemCom* and *NetCDF* are able to do this.

From the lower layer, it is expected that data access is fast, and should scale well for large data sets (i.e. more than 10 MB) and for many (i.e. more than 100000) large data sets. This will be examined in a series of benchmarks.

2.2 ADF

ADF is a hierarchical file manager, which is used by *CGNS*. An *ADF* database consists of a collection of *nodes* organised in a hierarchical way. Each node has a name, a label which defines its *role* in *CGNS*, array data of arbitrary size, a parent (except the root node), and an arbitrary number of child nodes. The time required for creating n nodes is proportional to $O(n^2)$. The only possibility of annotating nodes is by creating child nodes. *ADF* provides for accessing portions of the node data.

2.3 NetCDF

NetCDF is a file format and an programming interface for storing and retrieving arrays in an efficient manner (accessing portions of an arrays is possible). There is no hierarchical structure in a *NetCDF* dataset: A *NetCDF* dataset is like a UNIX file system directory containing several files (called *variables*). To add or delete a variable, the dataset has to be opened in the *define* mode. After that, the define mode has to be left. At that point, *NetCDF* does the necessary re-organisation in the database. As will be shown in the benchmarks, this is where *NetCDF* spends much time.

The *NetCDF* C API contains 74 functions. There is a C++ and a FORTRAN API as well.

NetCDF has currently a limit of 2GB per database, which is a bit small. The next release of *NetCDF* should solve this problem.

2.4 MemCom

Like *NetCDF*, *MemCom* has no hierarchical structure, it allows for accessing datasets down to the atoms of the set. Unlike *CGNS* and *NetCDF*, an arbitrary number of name/value pairs can be attributed to each dataset. These variables are stored as first-order relations in *descriptors* (not to be confused with *CGNS*-descriptors). Each dataset has a descriptor associated with it. Since descriptor and dataset are technically two different entities, variables may be added and deleted without affecting the performance. Such first-order relation can also be stored as data sets, a feature which is very important for storing unordered auxiliary information like physical constants, keywords, text, etc.

This is a very important feature since it eliminates the need of creating hundreds of datasets or nodes as it is the case in *ADF*/*CGNS* and *NetCDF*. The structure of the database becomes simpler, and from this the API of the upper layer benefits greatly.

In contrast to the current version of *NetCDF*, on platforms supporting large (i.e. 64bit) file systems, *MemCom* database sizes are only limited by the machine's capabilities.

Chapter 3

Performance Tests of *CGNS*, *MemCom*, and *NetCDF*

The performance of a “database” solution is a very important aspect in scientific computation and engineering. If not, everyone would make use of the wide range of available relational database management systems (such as Oracle, DB2, Informix, Sybase, MS SQL Server, PostgreSQL, MySQL, etc.), which are heavyweight and have an large overhead, but offer much more functionality than the systems presented here.

Instead of comparing *ADF* with *MemCom* and *NetCDF*, *CGNS* was compared with *MemCom* and *NetCDF*. The reason for this is because *CGNS* re-implements a significant part of *ADF*’s functionality on top of *ADF*, which affects performance.

3.1 Benchmark Programs

To assess the performance of the different solutions, i.e. *CGNS*, *NetCDF*, and *MemCom*, a test program was written for each system. The program takes the following command line arguments:

- The name of the database (in *NetCDF* dataset).
- The number of datasets (in *NetCDF* variables) to be created.
- The size of each dataset in number of 4-byte floating point numbers.

These programs create a new database, and in this database they write the given number of datasets. Each dataset is one-dimensional, with the specified size. The

datasets are written in one piece, i.e. in one logical access, since *CGNS* does not support I/O of portions of a dataset. This implies that, at the beginning of the execution, an array of the required size of floats is allocated and initialised. A pointer to this array is then given to the functions which write datasets.

For *NetCDF* and *MemCom*, the names given to the variables/datasets are `1`, `2`, `3`, and so on. In *CGNS*, the datasets are created as grid data for the X direction, each being in a different zone. The zones are named `1`, `2`, `3`, and so on. This of course implies an overhead creating the intermediary nodes, which gives *CGNS* a slight disadvantage in this benchmark. However, this "disadvantage factor" remains constant with the number of datasets created, and it will vanish quickly as the size of the datasets increases.

For *NetCDF*, the `nofill` option was set, which prevents double writing of the same location in the database (sometimes needed to avoid file fragmentation). Furthermore, no `nc_sync` call was being made. The maximum number of variables was increased in the header file in order to execute the tests. In the benchmark's source file, a `RESERVE` macro would define if all variables to be created would be defined at the same time, and *then* written, or if for each variable, the dataset would be put into `define mode`, the variable defined, then the dataset would be put into `data mode`, and the variable written. The version compiled with the `RESERVE` macro set is called *NetCDF-R*, while the other is simply called *NetCDF*.

For *MemCom*, the `dosync` parameter was set to `false`. All datasets are created and written atomically, by means of `mcDBputSet()`. This prevents datasets being written twice.

On *CGNS*, the datasets are created with a sequence of function calls to create a new zone and to write into that zone. All nodes are - from the perspective of the *CGNS* user - created atomically. *CGNS/ADF* does not flush pending data to disk before closing file descriptors, and does not provide any means to alter that behaviour.

All programs were written such that they would perform as best as possible for the given library. All libraries and programs were compiled with the same compiler and the same speed optimisations turned on (we used the C API's of the libraries).

3.2 Testing Environment

Test were performed on a dual *Pentium 500 MHz Linux box* with 512MB of RAM and a *SCSI* I/O facility and *ext2fs* file systems. The maximum database size was chosen to be a $\frac{1}{4}$ of the available RAM to avoid any swapping of memory. The *ext2fs* file system of Linux is of asynchronous type; this means that pending writes are generally not flushed to disk during program execution if the available memory is large enough to hold all writes. On synchronous file systems, when the program is terminated and all descriptors are closed, pending data for these descriptors is forced to disk. This means that program execution generally takes longer and the execution time reflects the total of operations performed on the database. On the other hand, since the flushing of pending data is only operating system-dependent, an asynchronous file system shows the differences between the different solutions more precisely. This is the reason why Linux was chosen as benchmark platform.

3.3 Create and Write One Dataset

In the first series of benchmarks, a single large dataset is created. This will show how well each system may take advantage of the available hardware and operating system facilities. The size of the dataset was up to 25'600'000 4-byte floating point numbers, giving a total size of roughly 100MB.

As can be seen from figure 3.3, all systems scale linearly with the set size. *MemCom* wins slightly over *NetCDF* and by a factor of two over *CGNS*. For such large datasets, the few extra nodes created in *CGNS* are negligible. *MemCom* is capable of making use of the optimal performance provided by the system. *CGNS* spends 50% of its time doing bookkeeping.

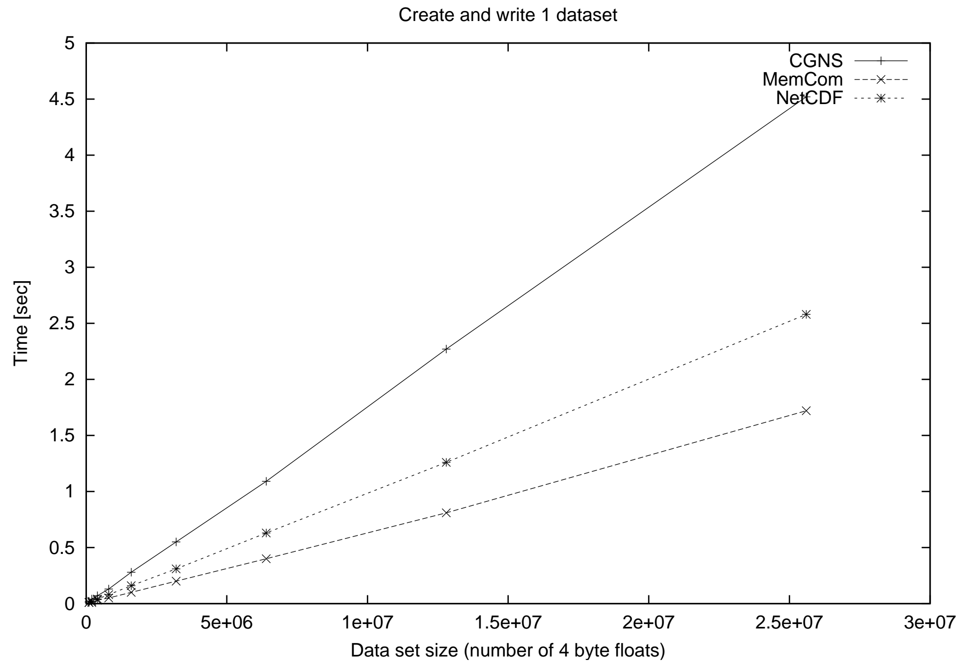

Figure 3.3: Create and write one dataset test timing.

3.4 Create and Write Many Datasets

Another series of benchmarks was performed to estimate the bookkeeping overhead for the creation of datasets/nodes/variables. For *CGNS*, the time could be divided by 5, since for each set of coordinates written, is has to create 5 nodes. However, we feel that this would not be fair to the other systems, because this is a direct result of the overuse of the hierarchical structure in *CGNS*. Furthermore, the following results show that it is irrelevant if this factor is taken into consideration.

In the benchmarks, databases with a given number of datasets/nodes/variables, all having the size of only 1000 floats (roughly 4KB of data), were created. To show the results over a wide range, the number of datasets would be increased at each step by a factor of two. For each step, three samples would be taken, with the best sample shown in figure 3.4:

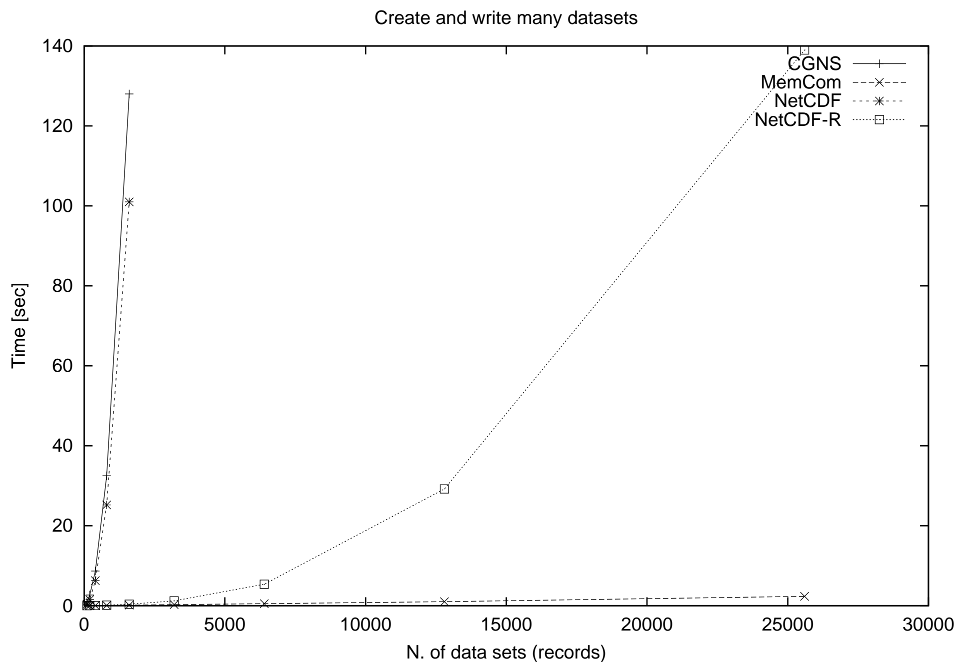

Figure 3.4: Create and write many datasets test timing.

The results in figure 3.4 clearly show that only *MemCom* scales well with the number of datasets. In fact, it scales linearly and performs - even for small numbers of datasets - always best. This is proof that the bookkeeping algorithms built into *MemCom* are of superior design. Whatever the nature of the problem may be, *MemCom* is always the fastest solution.

It is nevertheless interesting to analyse the results. *CGNS* shows a quadratic behaviour. This is due to the underlying *ADF* layer of *CGNS*, whose bookkeeping algorithms are not up to the job. Together with the excessive use of nodes in *CGNS* this explains why *CGNS*-based solutions are poor performers.

NetCDF-R seems to perform almost linearly at smaller sizes. It is interesting to note that the results are almost as good as those of *MemCom*. At a certain point however, the linear behaviour changes to a quadratic behaviour, quickly reaching overly long execution times. Still, for small to medium-sized problems, *NetCDF* could be a viable solution. The change to quadratic behaviour could be due to an limit which is reached. If this limit was increased, the linear behaviour could be preserved for a longer time. However there was no indication in the documentation or in the source code of *NetCDF* of such a limit and how it could be changed.

A problem with *NetCDF-R* is that the number of variables being created and the dimensions of each variable must be known in advance, which is seldom the case in finite element analysis, where different tools work in sequence on the same database (or dataset in *NetCDF*). However, if this is the case, one can define dimensions and variables once and then write the variables (*NetCDF-R*). *NetCDF*'s over-simple structuring of the database require all existing data to be copied to another place when new variables are added. This explains the quadratic behaviour of the non-reserving *NetCDF* program, which yields a bad performance similar to that of CGNS.

3.5 Mixed Benchmark

In this series, we write a moderate number of moderately sized datasets (100'000 floats each) to a database. The results are shown in figure 3.5:

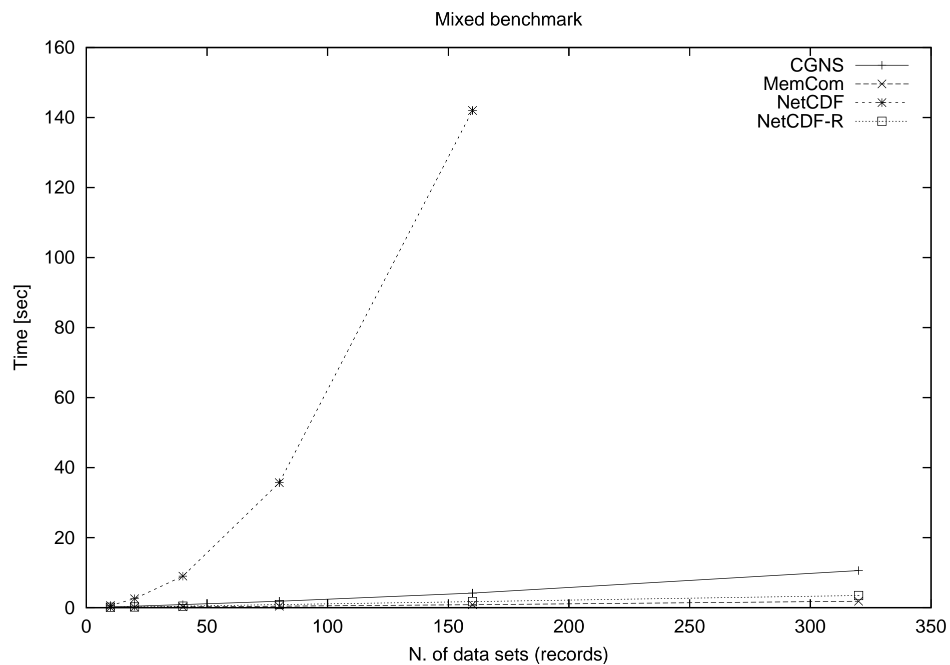

Figure 3.5: Mixed benchmark test timing.

As can be seen, the quadratic behaviour of the bookkeeping algorithms of *CGNS* and *NetCDF-R* is well masked by the time needed to write each dataset. For *NetCDF* the quadratic behaviour remains, since all data has to be moved when a new variable is inserted.

Chapter 4

The Upper Layer: Data Organisation and API

4.1 Requirements

The requirements pertaining to the upper layer are listed below:

- Data produced by the coupled CFD and CSM codes should reside in the same storage.
- A C++, a C, and a Fortran API should be available.
- The API should allow for compact code (to perform an operation with as little lines of source code as possible).
- The API should leave no room for ambiguities and delegate as little responsibility as possible to the programmer. It should be designed such that as much programming errors as possible are detected at compile time.
- The API and data storage should provide support for sub-domains (or branches, or zones, or partitions), while hiding the fact that the model is divided into different sub-domains from processors that do not support sub-domains. The ability of dividing a model into sub-domains is of interest in design and in the pre- and post-processing stages.

*NetCDF*does not really have an upper layer, i.e. a CFD or any other problem-oriented API must be written on top of *NetCDF*.

4.2 *CGNS*

The *CGNS* API uses *ADF* as file manager. Consequently, it inherits *ADF*'s problems of nodes having many children.

Sub-domains are written into different nodes called *zones*, which is problematic for codes that do not support sub-domains.

Since there are many different node types in *CGNS*, the *CGNS* API consists of 115 different functions. The overwhelming amount of node types comes from the lack of annotated data (in *NetCDF* called variables, in *MemCom* called items of a relational table). Instead, each variable must be stored in a separate node of its own type. This results in a considerable amount of different node types making the API (and its implementation) overly complicated.

The *CGNS* API makes an attempt to formalise access to data by providing data access functions for each type. Data entities are defined through the *SIDS*. Access to these entities is then provided by the *CGNS* API which implements the *SIDS*. Other data entities are not supported.

In addition, it is important to notice that reading and writing parts of node data is possible in *ADF*, but generally not possible when using the *CGNS* API: writing part of the data is not possible, and reading part of the data is allowed only for large data sets such as grid coordinates and flow solutions.

4.3 The *Julius* Data Structure and Database Definition

The *Julius* data manager has been designed to create portable and shareable data for heterogeneous environments and to create a common data representation and common data access functions. To achieve these goals the data access has been based on the *MemCom* data management system (see the documentation section in http://www.smr.ch). *MemCom* separates the logical and the physical structure of data, i.e. data are accessed symbolically, without knowledge of the position of the data on the storage media.

The common data representation has been defined in the *Julius* data structure definition (see http://www.smr.ch/Documentation/jtk/index.htm). Obviously, the common data representation has been tailored on one hand according to the requirements of the consortium and on the other hand to the specific data storage capabilities and functions offered by *MemCom*.

The common data access functions have been realised on two levels, an object oriented level and a functional level. The object oriented level offers two libraries: The *jtk* sequential data access classes and the jpmac parallel data access classes.

The *jtk* object oriented data access library has been designed for simplicity of the API. All data structures and the database and *MPI* calls are contained in the *jtk* classes. Thus, the application programmer can keep the code to a minimum. However, the data structures being defined and encapsulated in the *jtk* classes, the application programmer has no direct control over the data structure. The jpmac parallel data access library (meta-layer) does not encapsulate the data structures, thus leaving the full freedom of defining internal data strUctures to the apPlication programmer. The functional data access is supported by *MemCom* C API calls, the *MemCom* C API calls being sufficiently flexible to allow direct interaction with the database without intermediate software layers. An example of *MemCom* C API calls to the database is given in the Appendix.

A model contains information about the CAD model and the computational model(s). The CAD model is specific to the requirements of the *Julius* consortium, i.e. it is based on NURBS surfaces and lines and the boundary representation topology (see below).

CAD Models

The CAD meshes are based on the boundary representation (BREP) model, closely following the one defined by the *ACIS* modeler (http://www.spatial.com). The brep data structure and database definition consist of two parts, geometry and topology, geometry being represented by NURBS surfaces and line. Figure 4.3 shows the geometry and topology tree and the corresponding database tables.

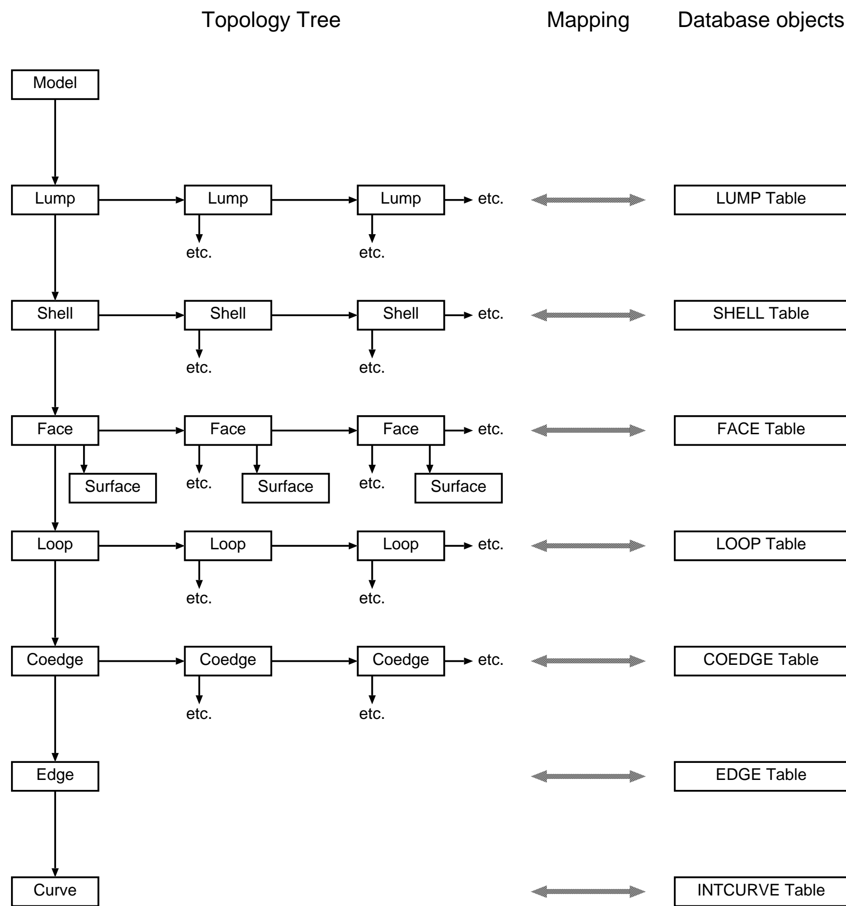

Figure 4.3: CAD objects in memory and their mapping to the database.

Computational Models

A computational model designates all data objects which together describe a numerical model for a particular type of analysis (see figure 4.3). Several computational models, each of them consisting of geometrical (mesh) and physical information, can be defined for a given model. The computational model data structure supports both single partition (block) and multi-partition (block) meshes. A partition (block) may consist of a structured or an unstructured mesh, and an unstructured mesh may contain one or more element types.

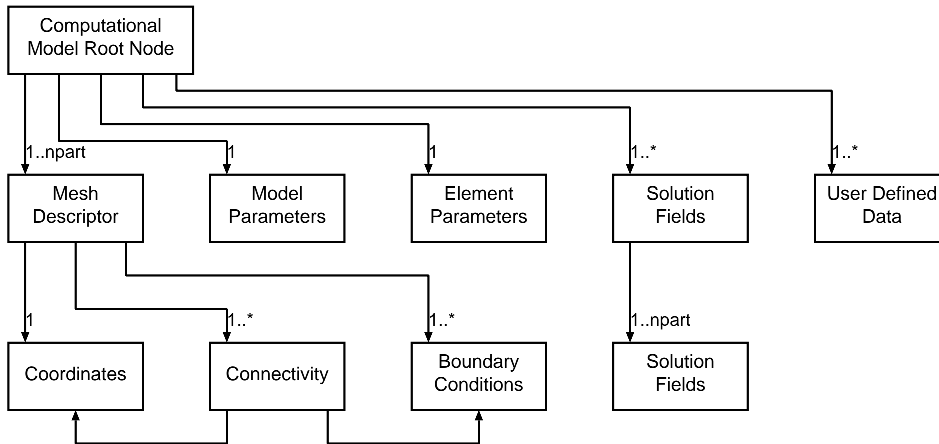

Figure 4.3: The data structure of the computational model.

The *jtk* Sequential Data Access Library

The data access functions provide the interface between the application programs and the *MemCom* data management system. Specifically, the data access functions provide for definition, insertion, extraction, and deletion of application-specific data residing on the common database. The *jtk* sequential data access library consists of a hierarchy of classes written in C++. Among the advantages of the object-oriented approach are dynamic manipulation of data, hiding of data base access functions and data structures, and the ability for extending the code without modifying the API.

The *MemCom* API allows for interfacing to C++/C and Fortran code (the Fortran application program interface requires that Fortran based applications should to be able to handle pointers).

Physically, data are stored in a UNIX file system directory with a number of subdirectories. Organising data in a directory tree does not necessarily mean that data are physically stored according to the same hierarchy, i.e. in the same file system or on the same disk.

Data pertaining to a specific combination of CAD geometry - computational model are collected in what we refer to as a *model* and stored in a directory created for this purpose. Thus, in addition to being stored separately without any connection to other models, a hierarchy of models can also be set up. This setup allows for great flexibility, while preserving simplicity by making use of the hierarchical structure of the file system. However, the file management of the complete database may not be left to the operating system tools alone, since it may result in

inconsistent data, i.e. if parts of the data base are removed by hand, overriding the global definition This is particularly true if, as a result of parallelisation of large meshes, i.e. for allowing parallel i/o, data are spread over many disks. Tools for file management, particularly for removing data from the data base are thus of primordial importance.

The *jpmac* Parallel Computational Model Data Access Library

The *jpmac* parallel computational model data access library consists of a layer (”meta-layer”) on top of the *MemCom* api calls to the database files. There are no parallel data access capabilities for the CAD database since there is no real need for it yet. *jpmac* assumes that a parallel program has been designed according to the *SPMD* (single program multiple data) parallel programming model and it makes use of the *MPI* message passing library: Copies of the parallel application program execute simultaneously on a parallel computer or in a cluster of tightly coupled computers, working on different portions of data, and exchanging data with *MPI* (see figure 4.3).

The basic idea of *jpmac* is on one hand to hide the message passing functions as well as the *MemCom* api and on the other hand - as opposed to the *jtk* library - to leave control over the data structure with the application.

Access to the data base is more or less synchronous, i.e. all copies of the application program request the same type of data at a given time interval. In the current version of *jpmac* the actual transactions with the data base are performed by processing element (pe) 0. An example is given in the appendix.

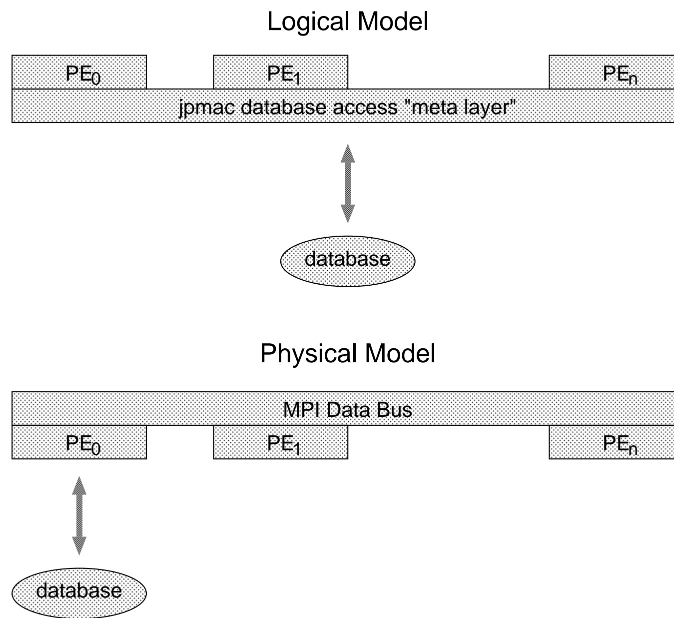

Figure 4.3: Logical and Physical Access to the database.

Chapter 5

Conclusion

The benchmarks performed clearly show that only *MemCom* scales well (linearly) with the size and the number of sets in a database. The quadratic behaviour of *CGNS* and *NetCDF* make them a lesser choice for simulations involving large unsteady problems. Therefore, from the point of view of performance, *MemCom* is the best choice.

It seems that, for the *CGNS* system, performance considerations during the design and the implementation have not been of primordial importance, as is shown with the second series of benchmarks. These benchmarks let *CGNS* create many nodes of different types, something which is quite realistic during non-stationary calculations.

NetCDF requires that all sets created during the lifetime of a database have to be defined a priori. Even then it does not scale with the number of sets in a database. Although *NetCDF* offers additional functionality by means of descriptive variables, *MemCom*'s associative arrays are more flexible and extensible.

Appendix A

Short Demonstration of the *MemCom* C API for *jtk*

In what follows a short demonstration of the database query by means of the *MemCom* api is given. A *MemCom* file containing the mesh of a computational model is opened and data are extracted.

Open a data base file *demo.db* in read-write mode:

```
int handle; /* db file handle */

if ((handle = mcDBopenFile("demo.db", "rw")) < 0)
    return mcErrStatus();
```

The function returns the file handle. All subsequent data access functions for `demo.db` will refer to `handle`. Close a data base file identified by `handle`:

```
mcDBcloseFile(handle);
```

The top-level computational model parameter table `MPAR` contains all necessary information to navigate further down in the mesh. `MPAR` is a relational table and has to be accessed by means of the table manager. Load the `MPAR` table and extract the relevant parameters (a function return value of `null` means that the table has not been loaded and `mcErrStatus` then returns the error code):

```
mcRTable *rt;

if ((rt = mcTBload(handle, "MPAR", 0)) == NULL)
    return mcErrStatus();
```

Extract parameters from the loaded table `rt`. The *array* version of the table extraction functions does not allocate the object dynamically. However, the object size has to be known in advance, which is trivial if the object size is 1. The example function below extracts a single `mcInt32` integer `npart` and places it in variable `npart`:

```
int n;
mcInt32 npart;

n = 1;
mctbextarrayi("npart", &npart, &n, rt);
```

Once all extraction operations are terminated the table can be released:

```
mcTBfree(rt);
```

To save current parameters to `MPAR`, load the table again

```
mcRTable *rt;

if ((rt = mcTBload(handle, "MPAR", 0)) == NULL)
    return mcErrStatus();
```

and replace the parameters

```
mcInt32 npart;

mctbinsi("npart", &npart, 1, rt);
```

and save back to the database

```
int status;

if ((status = mcTBstore(handle, "MPAR", 0)) != 0)
    return status;
```

Again, once all operations are terminated the table can be released:

```
mcTBfree(rt);
```

The computational mesh descriptor table `MDES.part` can now be loaded. As the mesh descriptor table is the root table of a partition's mesh it contains all information necessary for extracting the mesh of a partition. The data sets associated to the computational mesh descriptor are

```
MDES.part
COOR.part
GRID.part.ident
```

where `part` designates the partition number. Since the mesh descriptor table name consists of the generic name `mdes` and the partition number (partitions must be numbered consecutively 1,2,3...) the data set name of the mesh descriptor of partition 1 must be composed with `snprintf`:

```
char mdes[64];
int part;

part = 1;
snprintf(mdes, sizeof(mdes), "MDES.%d", part);
```

Next, load the computational mesh descriptor table and extract the mesh type `MESH`

```
char mtype[4];
n = 1;

if ((rt = mcTBload(handle, mdes, 0)) == NULL) /* Table not found */
   return mcErrStatus();

if (mcTBextArrayK("MESH", mtype, &n, rt) != 0) /* "MESH" key not found */
   return mcErrStatus();
```

In case of an unstructured mesh, i.e. `mtype[0]='U'`, read the mesh parameters pertaining to an unstructured mesh, i.e. read `NETYP`, i.e. number of element types:

```
mcInt32 netyp;
n = 1;

if (mcTBextArrayI("NETYP", &netyp, &n, rt) != 0)
    return mcErrStatus();
```

Read the array `LTYP` containing the `NETYP` element type numbers:

```
mcInt32 ltyp[100];

n = netyp;
if (mcTBextArrayI("LTYP", ltyp, &n, rt) != 0)
    return mcErrStatus();
```

When the table has been processed, release it:

```
mcTBfree(rt);
```

The data set `COOR.part` describes a set of coordinates pertaining to the partition `part`. To load it, compose the set name and load the array. Check first if the set exists (`mcDBinqSet`) and if so, load it by means of the read function `mcDBgetSet`.

```
mcFloat64 *coor;  /* array to store coordinates */
char label[64];   /* String to store data set name */

/* Compose name (no error check here!). Here: Partition 1 */
snprintf(label, sizeof(label), "COOR.%d", 1);

/* Check if set exists */
if (!mcDBinqSet(handle, label, &att)) {
    /* Yes, load it and get status in case read function fails */
    coor = mcDBgetSet(handle, label, att.size);

    /* Get status in case read function fails */
    if (!coor)
        return mcErrStatus();
} else {
    /* No, print error message and return... */
}
```

A similar procedure is used in case parts of the coordinate array shall be loaded. To extract the optional coordinate table descriptor information, load the descriptor

```
mcRTable *r = mcTBloadDesc(handle, label);

/* Extract here... */

mcTBfree(rt);
```

The descriptor can contain auxiliary information, like the storage format of the coordinate array (some people like to store coordinates by specifying all x coordinates, then all y coordinates, then all z coordinates - in that case the descriptor would contain a specific keyword). To store a coordinate array `COOR`

```
mcSize size;
mcFloat64 *coor;

/* set the size of the set */
size = ...;

/* allocate the coor array */
if ((coor = malloc(sizeof(*coor) * size)) == NULL) {
    perror("malloc");
    exit(EXIT_FAILURE);
}

/* initialise the coor array */
...

/* Compose name */
```

```
snprintf(label, sizeof(label), "COOR.%d", 1);

/* Store array */
if ((status = mcDBputSet(handle, label, "F", size, coor)) != 0)
    return status;

free(coor);   /* If not needed anymore */
```

To replace the coordinate table descriptor information (if the information has changed for some reason), load the descriptor

```
mcRTable *r = mcTBloadDesc(handle, label);

/*
 * Replace here...
 * and store the table back to DB
 */
mcTBstoreDesc(handle, label, r);
mcTBfree(rt);
```

The computational grid connectivity tables `GRID.part.cycle` describe various types of grids, referring to a set of coordinates or normals. The following sets are involved: `GRID.part.cycle`, `GRID-ATT.part.cycle` (if any), `COOR.part(.cycle)`, and `NORM.part(.cycle)` (if any).

To compose the set name, make use of the `snprintf` function. Example: compose name of the 3rd grid of the partition 2:

```
char mdes[64];

snprintf(mdes, sizeof(mdes), "GRID.%d.%d", 2, 3);
```

Check if the set exists (`mcDBinqSet`) and if so, load it with `mcDBgetSet`.

```
mcInt32 size = 0; /* size = 0 loads the whole set */
mcInt32 *Grid = mcDBgetSet(handle, label, size);
if (Grid == NULL)
    error...
```

Idem for the grid attribute table:

```
mcInt32 size = 0; /* size = 0 loads the whole set */
mcInt32 *GridAtt = mcDBgetSet(handle, label, size);
if (GridAtt == NULL)
    error...
```

A similar procedure is used in case parts of the grid shall be loaded. To extract the grid array table descriptor information, load the descriptor

```
mcRTable *r = mcTBloadDesc(handle, label);
```

and extract the grid attributes:

```
char *type = mcTBextK("TYPE", rt);
if (!strcmp(type, "VERTEX-ELEMENT") {
   /* Extract remainder information for this grid type */
} else if (!strcmp(type, "EDGE-ELEMENT") {
   /* do something with edge list stored in grid ... */
   etc...
}
```

Appendix B

A jpmac Parallel Access Example Program

This example program illustrates how the parallel data access classes work. The program opens at computational mesh database and then performs simple extract operations on parts of the computational mesh of the example database (see `to` and `from` variables below) before choosing the database. The operations occur in parallel, i.e. the program is launched `NPE` times, where `NPE` is the number of processing elements (or processes). Typically, the demo program would be launched by means of the `mpirun` procedure

```
mpirun -np ${NPROC} ./demo demo.db
```

where `demo` is the name of the program and `demo.db` the name of the example database.

```
// Demonstration Program for jpmac.
//
// The use of C++'s features here is very conservative.  Experienced C++
// programmers could simplify the code by structuring it a bit more.
// For example, The two-dimensional arrays (like 'coor', 'coora', etc.)
// could be replaced by a a Matrix class which stores all elements
// in a vector (necessary for fast I/O) and which overloads the '[]'
// operator.  The dataset names are created using 'snprintf', but could
// also be created using a 'strstream'.
//

#include <sys/types.h>

#include <stdio.h>
#include <stdlib.h>

#include <fstream.h>
#include <iomanip.h>
```

```
#include <iostream.h>

#include "mpi.h"

#include "mcDB.h"

#include "jpmacCModel.H"

ofstream out; // Output file "out.i", where i is the PE number [0..nproc-1]

// Exit message function
void
texit(jpmacCModel model, char *str, int status)
{
    out << " PE " << model.getMyPE() << " terminates at " << str
        << ", status=" << status << endl;
    out.close();
    mcExit();  // Terminate MemCom
    MPI_Finalize(); // Terminate MPI
    exit(1);
}

// prints out an array of floats
void
print_array(const mcFloat64* arr, int from, int dim1, int dim2)
{
    int k = 0;
    for (int i = 0; i < dim1; i++) {
        out << setw(5) << from + i;
        for (int j = 0; j < dim2; j++) {
            out << setw(12) << arr[k];
            k++;
        }
        out << endl;
    }
}

// prints out an array of integers
void
print_array(const mcInt32* arr, int from, int dim1, int dim2)
{
    int k = 0;
    for (int i = 0; i < dim1; i++) {
        out << setw(5) << from + i;
        for (int j = 0; j < dim2; j++) {
            out << setw(6) << arr[k];
            k++;
        }
        out << endl;
    }
}

int
main(int argc, char **argv)
{
    // Initialize library
    jpmacInit(argc, argv);

    // Declare a computational model.
    // Right now this must be done after MPI_Init, since the constructor
    // makes use of MPI (it reads the PE number).
    cerr << "Process starts: Create jpmacCModel" << endl;
```

```
jpmacCModel model;

// Open output file for this process.
// This must be done after constructor of jpmacCModel.

int mype = model.getMyPE();
char fname[8];

snprintf(fname, sizeof(fname), "out.%d", mype);
out.open(fname, ios::out, 0644);
if (!out) {
    cerr << "***ERROR Cannot open output file " << fname << endl;
    exit(-1);
}
out << "*** Process starts on PE " << model.getMyPE() << endl;

int status;

// Open computational model. Default is open for writing.
// Effective open only done in IO processor.
// Function loads the top-level model parameters (MPAR, EPAR, KEYS)
// OBS OBS keys not implemented!
// Function does not load partition parameters!

if ((status = model.open(argv[1])) != 0)
    texit(model, "model.open", status);
out << endl;
out << "Number of partitions    = " << model.getNumPartitions() << endl;
out << "Total number of nodes   = " << model.getNumNodes() << endl;
out << "Total number of elements = " << model.getNumElem() << endl;
model.getEpar()->show(out);

// Get the partition descriptor object jpmacCModelPartition of
// partition 1.
jpmacCModelPartition & partition = model.getPartition(1);

// Load partition parameters of partition 1 (there is only one...).
// Display partition parameters.

if ((status = partition.loadParameters(1)) != 0)
    texit(model, "partition.loadParameters", status);
partition.show(out);

// Extract selected coordinates to array coor and coordinate
// attributes (if any) to array coora. Print some of them for demo.

int from, to, nrow, ncol;
int nNodes = partition.getNumNodes();
int nDim = partition.getNumDim();

if (nNodes == 0 || nDim == 0)
    out << "No coordinates found" << endl;
else {
    from = 1;
    to = nNodes;
    mcFloat64* coor = new mcFloat64[(to - from + 1) * nDim];
    if ((status = partition.getCoor(from, to, &nrow, &ncol,
                                    coor)) != 0)
        texit(model, "partition.getCoor", status);
    out << "id=" << model.getMyPE() << " Some coordinates: " << endl;
    print_array(coor, from, 3, nDim);
    delete[] coor;
```

```
    int nNodeatt = partition.getNumNodeAtt();
    if (nNodeatt == 0)
        out << "No coordinate attributes found" << endl;
    else {
        mcInt32* coora = new mcInt32[(to - from + 1) * nNodeatt];
        if ((status = partition.getCoorAtt(from, to, &nrow, &ncol,
                                           coora)) != 0)
            texit(model, "partition.getCoorAtt", status);
        out << "id=" <<model.getMyPE()
            << " Some coordinate attributes: " << endl;
        print_array(coora, from, 3, nNodeatt);
        delete[] coora;
    }
}

// Extract topology objects and attributes (if any) to
// arrays topo and topoa. Print some of them.

int nTopo;
if ((nTopo = partition.getNumConnec()) == 0)
    out << "No topology objects found " << endl;
else {
    int nelem, nne;
    for (int cycle = 1; cycle <= nTopo; cycle++) {
        nelem = partition.getConnecNumRows(cycle);
        nne = partition.getConnecNumCols(cycle);
        if (nelem) {
            from = 1;
            to = nelem;
            mcInt32* topo = new mcInt32[(to - from + 1) * nne];
            if ((status = partition.getConnec(cycle, from, to, &nrow,
                                              &ncol, topo)) != 0)
                texit(model, "partition.getConnec", status);
            out << "id=" << model.getMyPE()
                << " Some topology values, cycle=" << cycle
                << " nelem=" << nelem << " nne=" << nne << endl;
            int k = 0;
            print_array(topo, from, 5, nne);
            delete[] topo;

            int nTopoAtt = partition.getConnecNumAtt(cycle);
            if (nTopoAtt == 0)
                out << "No topology attributes found " << endl;
            else {
                mcInt32* topoa = new mcInt32[(to - from + 1) * nTopoAtt];
                if ((status = partition.getConnecAtt(cycle, from, to,
                                                     &nrow, &ncol,
                                                     topoa)) != 0)
                    texit(model, "partition.getConnecAtt", status);
                out << "id=" << model.getMyPE()
                    << " Some topology attributes cycle=" << cycle
                    << " nelem=" << nelem
                    << " nTopoAtt=" << nTopoAtt << endl;
                print_array(topoa, from, 5, nTopoAtt);
                delete[] topoa;
            }
        }
    }
}

// Solution fields: Start by getting the field reference
```

```
    jpmacCModelField & field = model.getField();

    // Create a new descriptor
    const char *cn[5] = { "rho", "rhou", "rhov", "rhow", "rhoe" };
    field.create("Q", 3, "VERTEX", 5, cn);

    // Insert some dummy values in field
    double *val = new double[5];
    for (int i = 0; i < 5; i++)
        val[i]=(double)mype + 1;
    out << " app: ";
    for (int i = 0; i < 5; i++)
        out << val[i] << " ";
    out << endl;
    if ((status = field.set(1, val, mype+1, mype+1)) != 0)
        texit(model, "field.set", status);
    delete[] val;

    // Close computational model. Effective close only done in IO processor
    if ((status = model.close()) != 0)
        texit(model, "model.close", status);

    // Terminate
    jpmacExit();

    out << "Process terminates normally" << endl;
    out.close();
}
```

Appendix C

The Source Code of the MemCom Benchmark Program

```
/*
 * Test Program for MemCom-7
 *
 * (c) 2001 SMR Engineering & Development
 */

#include <sys/types.h>
#include <sys/times.h>

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "mcDB.h"

/*
 * This function returns the elapsed time (user + system time) in seconds.
 */
double
elapsed_time(void)
{
    struct tms clock;
    double usrtime;
    double systime;

    times(&clock);
    usrtime = (double) clock.tms_utime / (double)CLK_TCK;
    systime = (double) clock.tms_stime / (double)CLK_TCK;
    return(usrtime + systime);
}

/*
 * This function replaces ‘snprintf(buf, sizeof(buf), "%d", i)’.
 * The reason for it is that it is much faster than snprintf
 * (which could no small dataset sizes use too much CPU time).
 */
void
int_to_a(int num, char* s, size_t len)
```

```
{
    int i;
    int j;
    int k;

    for (i = 1; i * 10 <= num; i *= 10)
        ;
    k = 0;
    while (i > 0 && k < len - 1) {
        j = num / i;
        num %= i;
        i /= 10;
        s[k] = j + 48;
        k++;
    }
    s[k] = '\0';
}

int
main(int argc, char** argv)
{
    const char* dbname;
    int numsets;
    int numvalues;
    int i;
    char buf[30];
    int handle;
    float* values;
    double start_time;

    /* check args */
    if (argc != 4) {
        fprintf(stderr, "usage: %s name numsets setsize\n", argv[0]);
        exit(-1);
    }
    dbname = argv[1];
    numsets = atoi(argv[2]);
    numvalues = atoi(argv[3]);

    /* initialise the values array with bogus data */
    if ((values = malloc(sizeof(float) * numvalues)) == NULL) {
        perror("malloc");
        exit(-1);
    }
    for (i = 0; i < numvalues; i++)
        values[i] = i;

    /* measure the start time after allocation and initialisation */
    start_time = elapsed_time();

    /* initialise MemCom */
    mcInit();

    /* create database */
    if ((handle = mcDBopenFile(dbname, "rw")) == -1)
        exit(-1);

    /* create the required number of datasets and initialise them */
    for (i = 0; i < numsets; i++) {
        int_to_a(i, buf, sizeof(buf));

        if (mcDBputSet(handle, buf, "E", numvalues, values) < 0)
```

```
            exit(-1);
    }

    /* close the database */
    if (mcDBcloseFile(handle) < 0)
        exit(-1);
    mcExit();

    /* write out elapsed user and system time */
    printf("elapsed time: %f [s]\n", elapsed_time() - start_time);

    exit(0);
}
```

Appendix D

The Source Code of the NetCDF Benchmark Program

```
/*
 * Test Program for NetCDF
 *
 * (c) 2001 SMR Engineering & Development
 */

#include <sys/types.h>
#include <sys/times.h>

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "netcdf.h"

/*
 * This function returns the elapsed time (user + system time) in seconds.
 */
double
elapsed_time(void)
{
    struct tms clock;
    double usrtime;
    double systime;

    times(&clock);
    usrtime = (double) clock.tms_utime / (double)CLK_TCK;
    systime = (double) clock.tms_stime / (double)CLK_TCK;
    return(usrtime + systime);
}

/*
 * This function replaces ‘snprintf(buf, sizeof(buf), "%d", i)’.
 * The reason for it is that it is much faster than snprintf
 * (which could no small dataset sizes use too much CPU time).
 */
void
int_to_a(int num, char* s, size_t len)
```

```
{
    int i;
    int j;
    int k;

    for (i = 1; i * 10 <= num; i *= 10)
        ;
    k = 0;
    while (i > 0 && k < len - 1) {
        j = num / i;
        num %= i;
        i /= 10;
        s[k] = j + 48;
        k++;
    }
    s[k] = '\0';
}

void
handle_error(int status)
{
    fprintf(stderr, "netcdf error: %s\n", nc_strerror(status));
    exit(-1);
}

int
main(int argc, char** argv)
{
    const char* dsname;
    char buf[30];
    int dsid;
    int i;
    int dimid[NC_MAX_DIMS];
    int numvalues;
    int numvar;
    int oldmode;
    int start;
    int status;
    int *varids;
    float *values;
    double start_time;

    /* check args */
    if (argc != 4) {
        fprintf(stderr, "usage: %s name numvariables numvalues\n", argv[0]);
        exit(-1);
    }
    dsname = argv[1];
    numvar = atoi(argv[2]);
    numvalues = atoi(argv[3]);

    /* initialise the values array with bogus data */
    if ((values = malloc(sizeof(float) * numvalues)) == NULL) {
        perror("malloc");
        exit(-1);
    }
    for (i = 0; i < numvalues; i++)
        values[i] = i;

    /* allocate the varids array */
    if ((varids = malloc(sizeof(int) * numvar)) == NULL) {
        perror("malloc");
```

```
        exit(-1);
    }

    /* measure the start time after allocation and initialisation */
    start_time = elapsed_time();

    /* create a new dataset */
    if ((status = nc_create(dsname, 0, &dsid)) != NC_NOERR)
        handle_error(status);

    /* define dimensions */
    if ((status = nc_def_dim(dsid, "a", numvalues, &dimid[0]))
        != NC_NOERR)
        handle_error(status);
    if ((status = nc_def_dim(dsid, "b", 3, &dimid[1]))
        != NC_NOERR)
        handle_error(status);

    /* set no-fill mode for speed */
    if ((status = nc_set_fill(dsid, NC_NOFILL, &oldmode)) != NC_NOERR)
        handle_error(status);

    /* add many variables */
    for (i = 0; i < numvar; i++) {
        int_to_a(i, buf, sizeof(buf));

        /* define a new variable */
        if ((status = nc_def_var(dsid, buf, NC_FLOAT, 1, &dimid[0],
                                 &varids[i])) != NC_NOERR)
            handle_error(status);

#ifndef RESERVE
        /* go into dataset mode */
        if ((status = nc_enddef(dsid)) != NC_NOERR)
            handle_error(status);

        /* write the variable */
        start = 0;
        if ((status = nc_put_vara_float(dsid, varids[i], &start,
                                        &numvalues, values)) != NC_NOERR)
            handle_error(status);

        /* go into define mode */
        if ((status = nc_redef(dsid)) != NC_NOERR)
            handle_error(status);
#endif
    }

#ifdef RESERVE
    /* go into dataset mode */
    if ((status = nc_enddef(dsid)) != NC_NOERR)
        handle_error(status);

    /* write the variables */
    for (i = 0; i < numvar; i++) {
        start = 0;
        if ((status = nc_put_vara_float(dsid, varids[i], &start,
                                        &numvalues, values)) != NC_NOERR)
            handle_error(status);
    }
#endif
```

```
    /* close dataset */
    if ((status = nc_close(dsid)) != NC_NOERR)
        handle_error(status);

    /* write out elapsed user and system time */
    printf("elapsed time: %f [s]\n", elapsed_time() - start_time);

    exit(0);
}
```

Appendix E

The Source Code of the CGNS Benchmark Program

```
/*
 * cgns_test.c --
 *
 * Test Program for CGNS.
 *
 * (c) 2001 SMR Engineering & Development
 */

#include <sys/types.h>
#include <sys/times.h>

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "cgnslib.h"

/*
 * This function returns the elapsed time (user + system time) in seconds.
 */
double
elapsed_time(void)
{
    struct tms clock;
    double usrtime;
    double systime;

    times(&clock);
    usrtime = (double) clock.tms_utime / (double)CLK_TCK;
    systime = (double) clock.tms_stime / (double)CLK_TCK;
    return(usrtime + systime);
}

/*
 * This function replaces ‘snprintf(buf, sizeof(buf), "%d", i)’.
 * The reason for it is that it is much faster than snprintf
 * (which could no small dataset sizes use too much CPU time).
 */
```

```
void
int_to_a(int num, char* s, size_t len)
{
    int i;
    int j;
    int k;

    for (i = 1; i * 10 <= num; i *= 10)
        ;
    k = 0;
    while (i > 0 && k < len - 1) {
        j = num / i;
        num %= i;
        i /= 10;
        s[k] = j + 48;
        k++;
    }
    s[k] = '\0';
}

int
main(int argc, char** argv)
{
    const char* dbname;
    int numzones;
    int numvalues;
    int index_file;
    int index_base;
    int index_zone;
    int index_coord;
    int i;
    char buf[30];
    int isize[3][1];
    float *values;
    double start_time;

    /* check args */
    if (argc != 4) {
        fprintf(stderr, "usage: %s name numzones numvalues\n", argv[0]);
        exit(-1);
    }
    dbname = argv[1];
    numzones = atoi(argv[2]);
    numvalues = atoi(argv[3]);

    /* initialise the values array with bogus data */
    if ((values = malloc(sizeof(float) * numvalues)) == NULL) {
        perror("malloc");
        exit(-1);
    }
    for (i = 0; i < numvalues; i++)
        values[i] = i;

    /* measure the start time after allocation and initialisation */
    start_time = elapsed_time();

    /* create a CGNS database */
    cg_open(dbname, MODE_WRITE, &index_file);

    /* create base */
    cg_base_write(index_file, "Base", 1, 1, &index_base);
```

```
    /* initialise the isize parameter */
    isize[0][0] = numvalues;
    isize[1][0] = isize[0][0] - 1;
    isize[2][0] = 0;

    /* write numzones zones */
    for (i = 0; i < numzones; i++) {
        int_to_a(i, buf, sizeof(buf));

        cg_zone_write(index_file, index_base, buf, *isize, Structured,
                      &index_zone);
        cg_coord_write(index_file, index_base, index_zone, RealSingle,
                       "CoordinateX", values, &index_coord);
    }

    /* close CGNS file */
    cg_close(index_file);

    /* write out elapsed user and system time */
    printf("elapsed time: %f [s]\n", elapsed_time() - start_time);

    exit(0);
}
```