An XML based framework for self-describing parallel I/O data

András Belokosztolszki
University of Cambridge Computer Laboratory
JJ Thomson Avenue,
Cambridge CB3 0FD
United Kingdom
Andras.Belokosztolszki@cl.cam.ac.uk

Erich Schikuta
Institut für Informatik und Wirtschaftsinformatik,
University of Vienna
Rathausstr. 19/9,
A-1010 Vienna, Austria
schiki@ifs.univie.ac.at

Abstract

File I/O data is interpreted by high performance parallel/distributed applications mostly as a sequence of arbitrary bits.

This leads to the situation where data is 'volatile' information that has meaning only with the respective application responsible for it; a fact that contradicts the physical storage paradigm.

We claim that this situation must change and see the necessity to follow an approach – known from the database world – to interpret data as 'a model of reality'. To achieve this we add semantic knowledge to the data files.

Thus we propose an XML based language, which provides homogeneous framework for describing data on all interpretative levels from physical representation to logical information. In our framework physical files are consequently augmented by interpretative information specified in the proposed language.

This approach delivers beside semantic knowledge on the data the advantages of persistence and portability.

1. Introduction

In spite of a strong research focus in the last few years [7] file I/O in parallel or distributed applications remains a problematic issue until now. Very often we face the situation that in specific high performance applications the program code is changed on purpose to omit physical I/O. This is not only due to the I/O bottleneck and the missing I/O bandwidth to the external storage media but also to the fact that *persistence* is not an objective valued by this type of applications. File I/O in this context is mainly a necessity to extend the limited main memory of the physical hardware.

In typical supercomputing applications six types of I/O can be identified [5]:

1. input
2. debugging
3. scratch files
4. checkpoint/restart
5. output
6. accessing out-of-core structures

All these types of I/O handle the stored data, in one way or the other, as volatile information. It is only a sequence of bytes, which has a meaning only in the context of the program reading and/or writing the data. After the execution of the program this information is practically lost and the stored data is just garbage on the disk. Until now no standardized functional mechanism for technical and/or scientific data produced by high performance programs exist to maintain this information beyond the lifespan of the program. Only few (for example the ones in [3, 6]) and mostly very specific (like the one in [2]) approaches are proposed to overcome this situation until now.

We know that this will change in the future due to new and stimulating problems arising in biology, physics, etc., which will require new high performance applications with the need to store, administer and search intelligently gigantic data sets distributed over local and global storage medias. A good example to back our statement is the Online Analytical Processing in the Datagrid [9].

We will face a similar situation like in the well-known area of database systems, where data represents a model of the reality. It can be searched, analyzed, easily administered and at the same time data is efficiently at hand for arbitrary applications. This provides necessary means to express semantics in the data and consequently it raises a demand for mechanisms expressing semantics. Data has to be attributed with meta information describing the specific semantics of information in a standardized and processable way. This meta data enables applications to search the stored information intelligently.

However meta information in the context of high performance applications has to describe not only the logical knowledge within the data (semantic information) but also specific structural problem information of the parallel and/or distributed execution (*syntactical information*). Thus we propose an XML based language, which provides a homogeneous framework for describing data on all interpretative levels from physical representation to logical information (these levels are: data representation, structured file information, physical distributed data layout, problem specific data partitioning, and general information semantics). In our framework physical files are consequently augmented by interpretative information specified in the proposed language.

The layout of the paper is as follows. In the next section we present a novel file architecture defining the different level of file views. The XML-based PARSTORAGE language is introduced in section 3. This is followed by a comprehensive example of the application of the PARSTORAGE language. Here we present a proof of concept implementation of an application interpreting data attributed with information based on our approach. Finally we describe briefly our mechanism as a central part of a novel distributed file system and give an outlook on using our approach for information stored on the Datagrid.

2. A novel file hierarchy

The goal of a database system is to simplify and facilitate access to data [10]. This goal also holds adequately for data produced in high performance applications. We can learn from database systems to follow a similar approach to free the user from the burden of physical details. In database systems the above objective is achieved by the well-known three layered approach separating levels of different abstraction.

By keeping this model in mind we can define a similar approach for high performance data. To allow full flexibility for the programmer and the administration methods, we propose an architecture with three independent layers in the parallel I/O architecture (similar to the three-level-architecture of database systems):

Problem layer. Defines the problem specific data distribution among the cooperating parallel processes.

File layer. Provides a composed (canonical) view of the persistently stored data in the system.

Data layer. Defines the physical data distribution among the available disks.

These layers are separated conceptually from each other with mapping functions between these layers. *Logical data independence* exists between the problem and the file layer, and *physical data independence* exists between the file and

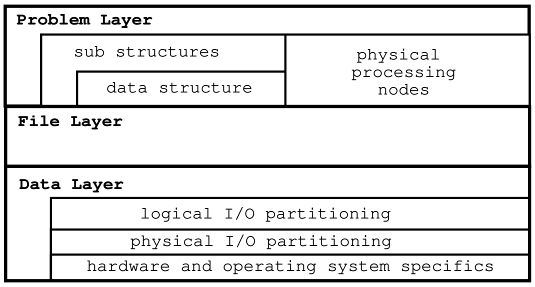

Figure 1. File hierarchy

data layer, which is analogous to the notation in database systems.

An even finer granularity of these layers is necessary to identify the different conceptual views for a typical parallel solution. These sub layers reach from pure logical information on the problem to physical representation of the data and comprise data representation, structured file information, physical distributed data layout, problem specific data partitioning, and general information semantics and so forth.

We will have a closer look on these layers (also depicted in figure 1) and sub layers and will show the instantiation by a specific example. We assume a typical, very simple, parallel HPF (High Performance Fortran) program writing high performance physics data (1000000 real values representing energy levels) to disk, which is administered by a parallel file system.

1. **Problem layer:** The problem layer represents the user's view on the problem specific data, which is expressed by the code of an HPF program.

   (a) **Data structure:** The first focus of interest is the data structure to write, which is only derived from the problem specification without any anticipation of the parallel execution flow. This data structure is basically a container type provided by the programming language. In practice (e.g. HPF) this is mostly a multi-dimensional array of an integral data type (e.g. *real*).

   Example: In our example this is a 2 dimensional array of 1000 times 1000 real values.

   (b) **Sub structures (SPMD derived):** The HPF language allows to specify a problem specific partitioning of the data within the SPMD framework, by defining a mapping of data structure elements to a logical processor array. This defines sub arrays, which are distributed accordingly to a defined distribution strategy (blocked, cyclic, etc.).

   Example: This phase splits the 2 dimensional array into 4 sub arrays in blocked (tiled) fashion and maps the fragments onto a 2 times 2 processor array.

(c) **Physical processing nodes:** The physical execution mode is defined by the numbers and types of real processors available for execution on the high performance hardware. This execution mapping defines mainly the execution flow of the program and consequently the data stream written/read to/from the underlying I/O environment.

Example: When the program is loaded only 2 physical processing nodes are available where 2 logical processors each are mapped respectively.

2. **File layer:** The file layer resembles the conceptual (logical) view of the data, which represents a "model of the real world". Basically it defines a sequential and structural (record information) form of the distributed data in a canonical normal form. This is the general view onto the stored information from any application, sequential or parallel.

   Example: In our example it gives the pure logical semantics of the data, e.g. it describes a sequence of real values representing energy levels of a HEP (high energy physics) experiment. From the programmers point of view it is the persistent mapping of the 2 dimensional FORTRAN array in column-major order onto a sequential file.

3. **Data layer:** The data layer is the lowest level of abstraction and describes how the data is actually stored. On this level the low-level data fragments are described.

   (a) **Logical I/O partitioning:** This level defines the partitioning of data supported by the underlying I/O subsystem mechanism. Basically it defines the mapping of main memory locations to storage locations (buckets) on the I/O media (buffer management).

   Example: Defines a I/O system specific view onto the data, e.g. striding. This could be done by the MPI-I/O view mechanism.

   (b) **Physical I/O partitioning:** The physical layout of the data (file fragments) onto the available storage media is defined by this level. It maps the information to store onto physical files on disks according to a specified fragmentation and layout which results into a set of files. This is system specific and dependent on the underlying software (file system) and hardware (storage media) architecture.

   Example: Delivers a set of physical files identified by unique file names, which can be accessed by the mechanism of the underlying file system.

   (c) **Hardware and operating system specifics:** Finally on the lowest level in our hierarchy this level defines the physical representation of the integral data information, which is dependent on the underlying software and hardware architecture.

   Example: This level defines how the real values of our data set are stored, e.g. big endian or little endian.

Consequently it is our goal to propose a framework, which allows to express the information on all layers of the presented file hierarchy.This led to the design of the `PARSTORAGE` language.

3. The `PARSTORAGE` language

In this section we introduce the PARSTORAGE language [1] based on XML, which makes it possible to add certain semantic information to a stored data set.

3.1. The `PARSTORAGE` Data Type Definition

To add semantic information to data files we have chosen XML (Extensible Markup Language). It is a language to describe languages, but unlike SGML (Standard Generalized Markup Language), which is much too general and complex, XML is simple, but still a powerful language. It has all the virtues of HTML (Hypertext Markup Language), but without HTML's limitations. As a direct consequence XML is well suited for annotating data with structural and semantic information. XML also enjoys a great popularity, which grows day by day, and due to its proliferation it can serve as a common interface between different applications.

The structure of our `PARSTORAGE` language is defined by a DTD (Document Type Definition). Its structure is organized as follows:

Processors give information about the logical processor structure. Several such processor structures can be given as different data arrays can be distributed over different logical processor layouts.

Type information to describe the data stored. It supports elementary types as well as composite types like structures or arrays.

Alignment information to cater for optimized data distribution.

Distribution information for mapping the logical file elements onto parts of physical files.

The relevant portion of the PARSTORAGE DTD is presented next to illustrate the above outlined structure:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ELEMENT PARSTORAGE
   (PROCESSORS*, TYPE+, ALIGN*, BLOCK+)>
<!ATTLIST PARSTORAGE
   VERSION CDATA #REQUIRED>
<!ATTLIST PARSTORAGE
   TIMESTAMP ID #REQUIRED>
```

Note, that version information is included as well as timestamps to follow the temporal change of both the logical (i.e. data changes) and physical files (i.e. different data distributions). This is imperative to cater for smart and dynamic physical file re-distribution like the one of ViPIOS [8].

3.2. Semantic Information

Basically we distinguish between pure semantic information, which comprises the problem and file layer and distribution information describing the data layer.

The semantic information is made up of the following expressive modules, which are defined by the respective DTD parts.

- *The different processor grids the data is distributed onto.* These processor grids are described by an optional name, the number of dimensions of the grid, and the extent of each dimension.

```
<!-- processors -->
<!ELEMENT PROCESSORS
   (PROC_DIMENSION)+>
<!ATTLIST PROCESSORS
   NAME CDATA #REQUIRED>

<!ELEMENT PROC_DIMENSION EMPTY>
<!ATTLIST PROC_DIMENSION
  LOWER CDATA "1">
<!ATTLIST PROC_DIMENSION
  UPPER CDATA #REQUIRED>
```

  These are referring to logical processor layouts needed for mapping data onto them. These logical processors are mapped to physical ones by the execution environment. Because of this, several logical processor layouts can be specified for the same logical file, one for each complex data type.

- *The data types stored in the logical file.* This section consists of a sequence of the types which are stored in the logical file, and in the same order as they are stored in the logical file. To achieve a more concise description, and to follow more closely the data types of the stored data, constructors help to build up complex data types. These constructors are *arrays* and *structures*. Knowing the size of each type, the information about the types stored enables applications to tell the type of any byte in a logical file.

```
<!-- hpf data structure -->
<!-- Intrinsic Data Types -->
<!ELEMENT TYPE (ETYPE|ARRAY|TYPE)+>
<!ATTLIST TYPE TYPENAME CDATA #IMPLIED>
<!ATTLIST TYPE NAME CDATA #IMPLIED>

<!ELEMENT ETYPE EMPTY>
<!ATTLIST ETYPE TYPE CDATA #REQUIRED>
<!ATTLIST ETYPE LENGTH CDATA #REQUIRED>
<!ATTLIST ETYPE NAME CDATA #IMPLIED>

<!-- Arrays -->
<!ELEMENT ARRAY (TYPE, DIMENSION+)>
<!ATTLIST ARRAY NAME CDATA #IMPLIED>
<!ATTLIST ARRAY
   MAJOR (ROW|COLUMN) "ROW">
<!ATTLIST ARRAY DISTRIBUTE_ONTO
   CDATA #IMPLIED>
<!ELEMENT DIMENSION EMPTY>
<!ATTLIST DIMENSION LOWER CDATA "1">
<!ATTLIST DIMENSION
   UPPER CDATA #REQUIRED>
<!ATTLIST DIMENSION DISTRIBUTE
   (BLOCK|CYCLIC|NO) #IMPLIED>
<!ATTLIST DIMENSION
   DIST_SKALAR CDATA "1">
```

- *Some align information.* Align information as in HPF describes a recommendation how data should be distributed onto processors/nodes.

```
<!-- Alignment -->
<!ELEMENT ALIGN EMPTY>
<!ATTLIST ALIGN WHAT CDATA #REQUIRED>
<!ATTLIST ALIGN WITH CDATA #REQUIRED>
```

3.3. Distribution Information

Distribution information is organized in a similar way as it is done in MPI-IO.

Blocks specify regions in the logical file. A physical file is characterized by a sequence of such blocks. The byte order in the physical file is the same as the byte order within a block. After the bytes of a block the bytes of the next block follow.

The most basic type of blocks is a BYTEBLOCK which represents a single byte. A more complex block is specified by four attributes and a child block, that itself can be a complex block. The four attributes are OFFSET, REPEAT, COUNT, STRIDE and SKIP.

```
<!-- data distribution in this file -->
<!ELEMENT BLOCK (BLOCK|BYTEBLOCK)+>
<!ATTLIST BLOCK OFFSET CDATA #REQUIRED >
<!ATTLIST BLOCK REPEAT CDATA #REQUIRED>
<!ATTLIST BLOCK COUNT CDATA #REQUIRED>
<!ATTLIST BLOCK STRIDE CDATA #REQUIRED>
<!ATTLIST BLOCK SKIP CDATA #REQUIRED>
<!ELEMENT BYTEBLOCK EMPTY>
```

To explain the meaning of the above attributes let us suppose, that we have already defined a child block, which is a BLOCK – in the most simple case BLOCK is just a byte, i.e. BYTEBLOCK. Let the size of the child block be 5 bytes. The current block is built up of this child block as in Figure 2.

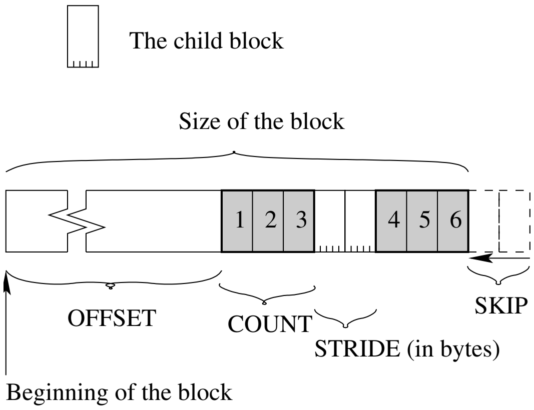

Figure 2. An example block

The OFFSET is the number of **bytes** from the beginning of the block till the first child block.

The COUNT specifies the number of contiguous child blocks. In our case it is 3.

The STRIDE is the number of **bytes** between the contiguous groups of COUNT number of child blocks. In this case it equals 10.

The REPEAT is the number contiguous groups of COUNT number of child blocks. In this case it is 2.

The SKIP is a signed number. It gives the number of **bytes** we need to add to $(COUNT * size\_of\_child\_block + STRIDE) * REPEAT + OFFSET$ to get the size of the current block. In the current case it is -10.

The contents of a physical file can be described as a block. In this case the OFFSET will be the same as the address in bytes from the beginning of the child block.

A block may also contain a sequence of child blocks. Note that the mapping of the bytes in the logical file is *not monotonic*. By using the above blocks any mapping of the bytes in the logical file can be specified.

A simple block example, where a bloc contains one contiguous block of 250 child block from the 2000^{th} byte, and the child block contains four contiguous bytes is shown next:

```
<BLOCK OFFSET="1000" REPEAT="1"
       COUNT="250" STRIDE="0" SKIP="2000">
  <BLOCK OFFSET="0" REPEAT="4"
         COUNT="1" STRIDE="0" SKIP="0">
    <BYTEBLOCK/>
  </BLOCK>
</BLOCK>
```

4. An example and a proof of concept application

In this section we will present a full example with a complete XML description. These XML files are used by ParXML [1], the proof of concept application, which is presented in the following.

We assume a file described by a respective HPF program, which contains an integer*1 array with 1000 elements. The array is distributed onto 4 processors in a cyclic(2) way. A physical file is assigned to each processor, and each physical file contains the portion of the array stored by its assigned processor. Each of the 4 physical files has a PARSTORAGE description attributed. This information can be part of the physical file – for example MPI-IO supports header information in physical files – , or can be stored separately either next to the physical file location or in a database.

Fore simplicity we present the full description for the first file only (which, once again to keep the example simple, is assumed to be mapped onto the first processor):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE PARSTORAGE SYSTEM
   "parastorage.dtd">
<PARSTORAGE VERSION="0.9" TIMESTAMP="and1">
  <PROCESSORS NAME="proca">
    <PROC_DIMENSION UPPER="4"/>
  </PROCESSORS>

  <TYPE>
    <ARRAY NAME="a" DISTRIBUTE_ONTO="proca">
      <TYPE>
        <ETYPE TYPE="INTEGER" LENGTH="1"/>
      </TYPE>
      <DIMENSION UPPER="1000"
                 DISTRIBUTE=CYCLIC"
                 DIST_SCALAR="2"/>
    </ARRAY>
  </TYPE>
```

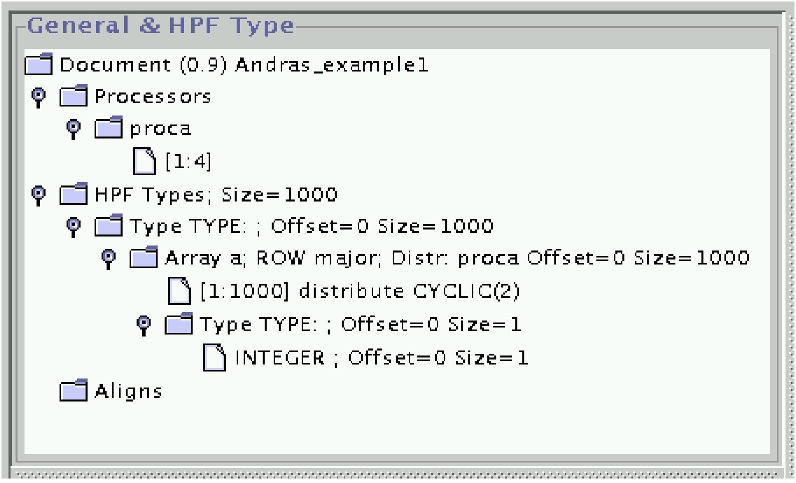

Figure 3. Semantic information

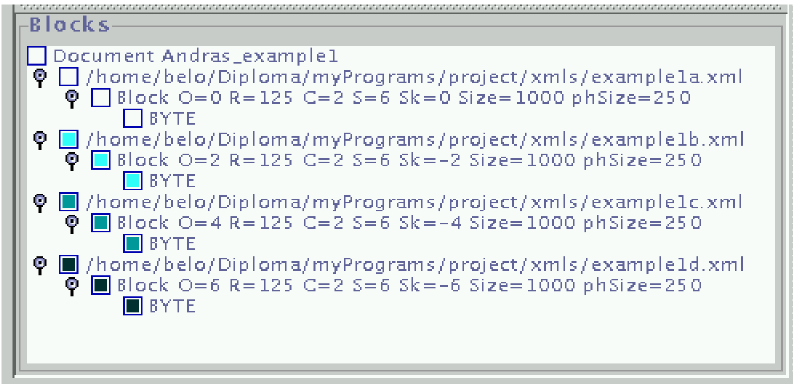

Figure 4. Distribution information

```
  <BLOCK OFFSET="0" REPEAT="125"
         COUNT="2" STRIDE="6" SKIP="0">
    <BYTEBLOCK/>
  </BLOCK>
</PARSTORAGE>
```

For the three other files the difference is only within the BLOCK statement, which is for the respective files:

Second file

```
...
<BLOCK OFFSET="2" REPEAT="125"
   COUNT="2" STRIDE="6" SKIP="-2">
...
```

Third file

```
...
<BLOCK OFFSET="4" REPEAT="125"
   COUNT="2" STRIDE="6" SKIP="-4">
...
```

Fourth file

```
...
<BLOCK OFFSET="6" REPEAT="125"
   COUNT="2" STRIDE="6" SKIP="-6">
...
```

The information of the 4 files (which is simply stored together with the files) allows the easy reconstruction of the original file out of the fragments. For a proof of concept we implemented a Java program ParXML [1], which reads the PARSTORAGE information, processes it and visualizes it graphically. It further can reconstruct the original file and produces the canonical form. The ParXML program extracts the semantic information (see Figure 3), analyzes the distribution information (see Figure 4), and visualizes the logical file in canonical form (see Figure 5).

5. Conclusion and future work

We presented PARSTORAGE [1], an XML language for storing meta information for parallel I/O files. This frame-

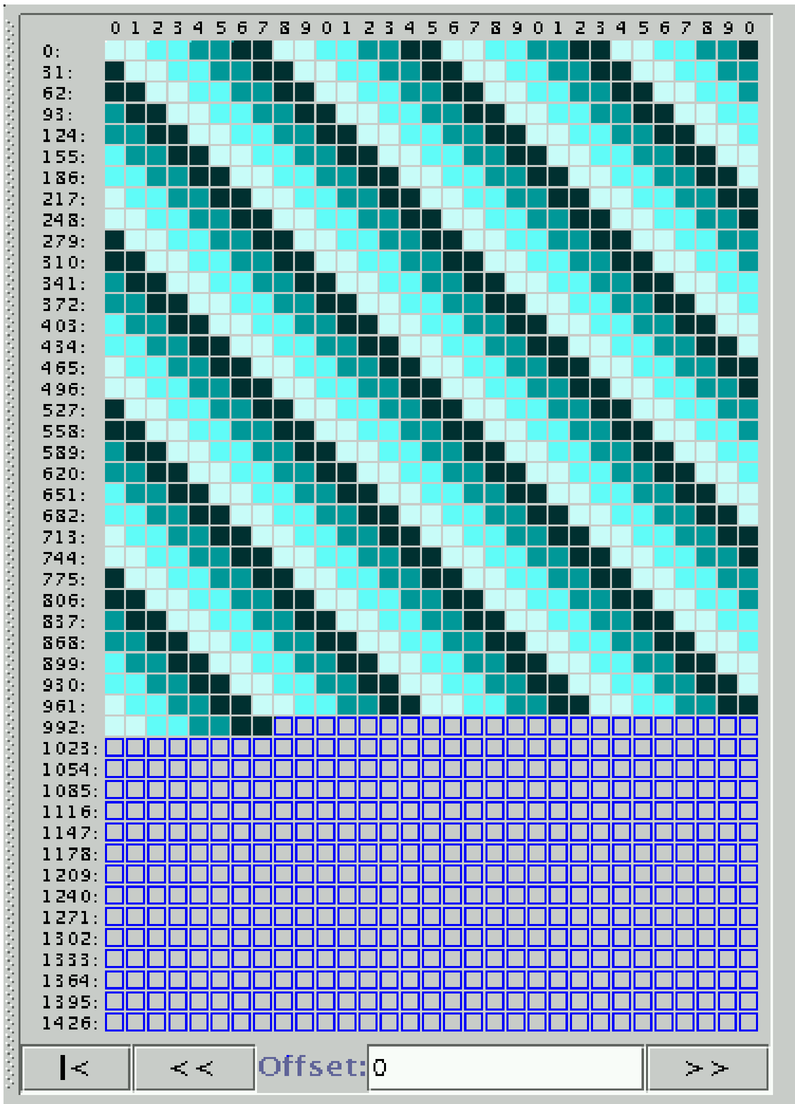

Figure 5. Visualization of the logical file

work allows to express semantic information attributed with the files and delivers the property of persistence for parallel I/O data.

In the last few years at the University of Vienna the ViPIOS system was developed [8], which is a client server based I/O system for parallel and distributed applications. Work is on the way to extend this system to deliver a viable platform for Grid applications, storing huge data sets distributed over dislocated storage resources. Part of ViPIOS is a file system for administering distributed files transparently for the user. The proposed XML approach acts in the system in two ways; on one hand it provides a user interface allowing to specify the layout of the file, on the other hand it is the expressive mechanism within the system to administer the distribution information of the files stored in the file system across several sites on the Grid. For this reason the `PARSTORAGE` language was extended to catch the specifics of a Grid infrastructure [4]. However here shows the XML approach its quality and justification due to the inherent and natural possibility of XML to be extended at will.

Acknowledgement

The work described in this paper was partly supported by the Special Research Program SFB F011 AURORA of the Austrian Science Fund. András Belokosztolszki also gratefully acknowledges the support of the Socrates-Erasmus exchange program.

References

[1] A. Belokosztolszki. An XML based language for meta information in distributed file systems. Master's thesis, University of Vienna and Eötvös Loránd University of Science, Budapest, 2000.

[2] N. Bhatti, J.-M. L. Goff, W. Hassan, Z. Kovacs, P. Martin, R. McClatchey, H. Stockinger, and I. Willers. Object serialisation and deserialisation using xml. In *10th International Conference on Management of Data (COMAD 2000)*, Pune, India, Dec. 2000.

[3] D. G. Feitelson and T. Klainer. *XML, Hyper-media, and Fortran I/O in [7]*. 2001.

[4] R. Felder. ViPFS: An XML based distributed file system for cluster architecture. Master's thesis, University of Vienna, Vienna, 2001.

[5] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471. IEEE Computer Society Press, 1993. Reprinted in the book "High Performance Storage and Parallel I/O" (http://www.buyya.com/superstorage/, 2001, pages 539–547).

[6] The NCSA HDF home page. http://hdf.ncsa.uiuc.edu/.

[7] B. Rajkumar, J. Hai, and C. Toni. *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. John Wiley and Sons, 2001.

[8] E. Schikuta, T. Fuerle, and H. Wanek. ViPIOS: The Vienna Parallel Input/Output System. In *Euro-Par'98*, Sept. 1998.

[9] B. Segal. Grid computing: The european data project. In *IEEE Nuclear Science Symposium and Medical Imaging Conference*, Lyon, Oct. 2000.

[10] A. Silberschatz, H. Korth, S. S., H. F. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, 1996.