ViMPIOS, a "Truly" Portable MPI-IO Implementation*

Kurt Stockinger†, Erich Schikuta
Institute for Applied Computer Science and Information Systems
Department of Data Engineering, University of Vienna,
Rathausstr. 19/4, A-1010 Vienna, Austria

Abstract

We present ViMPIOS, a novel MPI-IO implementation based on ViPIOS, the Vienna Parallel Input Output System. ViMPIOS inherits the defining characteristics of ViPIOS, which makes it a client-server based system focusing on cluster architectures. ViMPIOS stands out from all other MPI-IO implementations by its "truly" portable design, which allows not only applications to be transferred between parallel architectures easily but also to keep their original performance characteristics on the new platform as far as possible. This is kept by the "smart" AI-Blackboard module of ViPIOS, which is responsible for an appropriate data layout. Specifically in this paper we concentrate on the algorithm, which maps MPI-IO data structures on respective ViPIOS structures, and thus allows to exploit the ViPIOS properties.

1 Introduction

Over the last few years the so-called I/O bottleneck turned out as the limiting factor in high-performance computing. Thus, the performance of parallel systems in many applications is not bound by the number of available processors, but by the bandwidth of the I/O subsystem.

Therefore, the research on parallel I/O turned out as an important area in the high performance computing community. An important contribution is MPI-IO as part of the MPI standard [4], which supports parallel file accesses analogously to the MPI message passing scheme. It allows non-contiguous file access patterns and supports collective disk operations.

At the moment four different MPI-IO implementations are available, namely ROMIO [7], PMPIO [2], HPSS [9], and MPI-IO/PIOFS - an MPI-IO implementation by IBM Watson Research Center specific for the IBM SP-2 architecture. The predominant MPI-IO implementation is ROMIO, which is also part of the MPICH library, the most popular MPI implementation available for the widest spectrum of parallel systems.

One design goal of MPI (and thus MPI-IO) was portability, which allows the migration of applications from one parallel architecture to another one without loosing performance drastically.

However, hereby all available systems show a general weakness. They can not guarantee portability in the sense of the above definition. They allow the migration of applications between different systems, but they can not guarantee the same performance characteristics.

In this paper we present a novel MPI-IO implementation, ViMPIOS (Vienna Message Passing/Parallel Input Output System), a portable, client-server based MPI-IO implementation, which tackles the performance problem of the other systems.

ViMPIOS is based on ViPIOS, a fully-fledged parallel I/O runtime system, which was developed focusing on cluster systems. The main characteristics of ViPIOS are its client-server architecture and a smart AI-based file distribution mechanism.

The data layout defined by the programmer is used to determine a near optimal disk layout specific for the underlying I/O architecture. This allows high performance not only in dynamically changing environments, e.g. multi-user and multi-tasking situations typical in cluster systems, but also for the migration of applications form one system to another one with different I/O architectures.

This is reached in ViPIOS by the introduction of an additional data abstraction layer. Conventionally, the physical data layout on disk is defined by the programmer via MPI-IO data types in the application program. It can be further be tuned by specific hints, which try to catch the underlying storage architecture. However, if the application is used on other hardware systems, it has to be ported by hand specifically according to differentiating I/O characteristics (as number of disks, disk characteristics, bus systems etc.) to guarantee the expected performance. Here ViMPIOS shows its strength. Due to the smart AI layout apparatus in ViPIOS the adaptation to a new system is auto-

*This work was carried out as part of the research project "Language, Compiler, and Advanced Data Structure Support for Parallel I/O Operations" supported by the Austrian Science Foundation (FWF Grant P11006-MAT)

†Now at IT Division, CERN, CH-1211 Geneva 23, Switzerland. Email: Kurt.Stockinger@cern.ch

matically done within the ViPIOS system transparently to the programmer. Therefore, ViMPIOS, the MPI-IO layer of ViPIOS, is the only MPI-IO implementation so far, which provides full portability from one system to another one without additional program modifications and without loosing performance.

In the first section of the paper the data abstraction mechanism in ViPIOS is described. The second section gives with a short description of MPI-IO and what is actually realized in ViMPIOS. Later, we elaborate on the design and implementation of advanced derived data types which are typical for MPI-IO. We point out the extended flexibility of ViPIOS' internal representation of derived data types and we also show that besides standard MPI-IO data types even more complex description mechanism are possible. To round up our approach we describe a typical MPI-IO application program and justify our approach by a performance comparison to ROMIO.

2 Data Abstraction in ViPIOS

Generally speaking the distribution of the data in focus of a high-performance application is defined by the high level distribution mechanism of high-performance languages or library interfaces. In most systems which support parallel I/O this data distribution information builds the basis for the physical data layout on disks. This approach is found in all conventional MPI-IO implementations, which endangers the portability principle of high-performance applications. Such an effect is a result to different I/O characteristics, e.g. number and performance of disks, bus systems and many more, and situations with changing and/or not foreseeable system characteristics, e.g. the workload in multitasking environments (typical for cluster systems). In order to characterize parallel I/O efficiently, we define two issues which have to be addressed: *static fit property* and *dynamic fit property*.

The static fit property tries to adapt the disk access profile of the program according to the necessities of the problem specification or the programming paradigm (typical for the Single Program Multiple Data, SPMD, approach). This can be realized by the programmer or the compiler. For example, a specific distribution of the application is resembled by a corresponding data layout on disks.

The dynamic fit property denotes the adaptability of the I/O runtime subsystem to application's and/or environment's characteristics which could not be foreseen at compile time.

We see a solution to this problem in a combination of both approaches, which is a dedicated, smart, concurrent executing runtime system, gathering all available information on the application process both during compilation and the runtime execution.

This approach is followed in ViPIOS, the Vienna Parallel Input Output System [5]. ViPIOS is an I/O runtime system which provides efficient access to files by optimizing the data layout on the disks and allowing parallel read/write operations. ViPIOS is based on a database approach and supports high-performance languages (as HPF [3] and the VFC [1]), standard interfaces (MPI-IO) and persistent files.

The data administration approach in ViPIOS follows the data independence principle known from database systems. Adapted to a parallel I/O system this means that the physical data layout on disk is not defined by the problem layout of the application program and the programmer's hints alone.

Central part of ViPIOS is the *Fragmenter* [8], a kernel module, which is responsible for data layout decisions and the dynamic data distribution of the application data to the disks. The Fragmenter is based on a mathematical cost model describing the performance profile of the underlying hardware and applies a combination of an A^* algorithm and AI-blackboard system, which represents the expert knowledge about data layout, message passing, and buffer management. This approach is comparable to the way followed in the Panda system [6], where also unconventional methods (neural networks, fuzzy sets, genetic algorithms and simulated annealing) are applied to deal with uncertain or incomplete knowledge.

Generally two independent data layers exist, the logical or *Problem* layer and the physical or *Disk layer*:

- The Problem layer defines the problem specific data distribution among the cooperating parallel processes.
- The Data layer defines the physical data distribution among the available disks.

These layers are separated conceptually from each other. The distribution of the data in the problem layer is defined by the programmer by the data distribution mechanism of his programming paradigm, e.g. the data type structures in MPI. The Fragmenter defines the data layout in the Data Layer and provides also the mapping functions between the Problem and the Data Layer. The concept is depicted in Fig. 1.

This functionality in ViPIOS allows a totally new approach to the development of a dynamically adapting MPI-IO implementation. Basis for this approach is an algorithm which maps the MPI data structures to the proprietary ViPIOS structure. This algorithm is defined in the following sections.

3 Derived Data Types in MPI-IO

In this section we give a short overview of MPI-IO derived data types. However, rather than giving detailed information of the MPI-IO syntax, we will elaborate on a formalized description of a typical MPI-IO application which

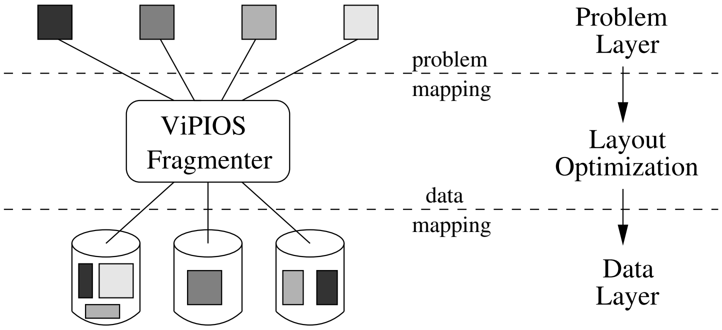

Figure 1. Data abstraction.

we will use as the basis for our further analysis throughout the rest of this paper. Later on we will present the ViPIOS Distribution-Mapping-Algorithm which, on the one hand, accomplishes the implementation of MPI-IO derived data type features, and, on the other hand, formalizes the mapping mechanism to the underlying ViPIOS, the actual ”brain” of ViMPIOS.

We start with a short algorithm on a typical MPI-IO program with a file view based on a derived data type. The single steps are:

- Step 1: Specification of the data file to be accessed, e.g. 2-dimensional file array with 6x4 elements.
- Step 2: Specification of the distribution and the corresponding distribution arguments, e.g. CYCLIC(2), CYCLIC(1).
- Step 3: Creation of the processor grid, i.e. number of processors of each dimension of the data file, e.g. 2x2 (2 processors per dimension).
- Step 4: Integration into the derived data type, e.g. MPI_Type_create_darray.
- Step 5: application of the newly created data type in the file view.

The whole algorithm is summarized graphically in Fig. 2.

4 ViMPIOS Distribution-Mapping Algorithm

On giving a formalized specification of an MPI-IO program, we can now deduce an algorithm which describes the distribution and the mapping of data onto the ViPIOS internal data structures. Again, we will choose a graphical approach to discuss how the MPI-IO features are implemented in ViPIOS such that the strength of ViPIOS, namely portability and dynamic user-independent optimized data distribution are still exploited.

The ViMPIOS Distribution-Mapping Algorithm consists of the following four basic phases:

- Phase 1: Distribution Analysis
  - Simple Case
  - Complex Case
- Phase 2: Projection
- Phase 3: Integration
- Phase 4: Mapping onto ViPIOS

Let us now briefly describe each step of the algorithm. First, in the *Distribution phase*, the data file which is to accessed has to be analyzed. Thus, assume a one-dimensional file of 4 elements and a processor grid with 2 processors. Further assume a CYCLIC(2) data distribution, which means that each processor gets 2 elements. We call this a *simple case* since each processor has the same workload. In contrast, the so-called *complex case* is considered as the case where at least 2 processors have a different workload. This is true, for example, if the file consists of 6 elements rather than 4. Thus, processor 0 would get 4 elements whereas processor 1 merely gets 2.

This distribution analysis has to be done for each dimension of the file. We refer to this step as the *Projection phase*. Taking a look at the introductory example yields a complex case for the dimension 0 - since 6 elements are distributed onto 2 processors according to CYCLIC(2) - and a simple case for the dimension 1 (since 4 elements must be distributed onto 2 processors).

During the *Integration phase*, the results from the projection of each dimension must be combined into one derived data type.

Let us apply this algorithm to processor 0. The graphical analysis is given in Fig. 3.

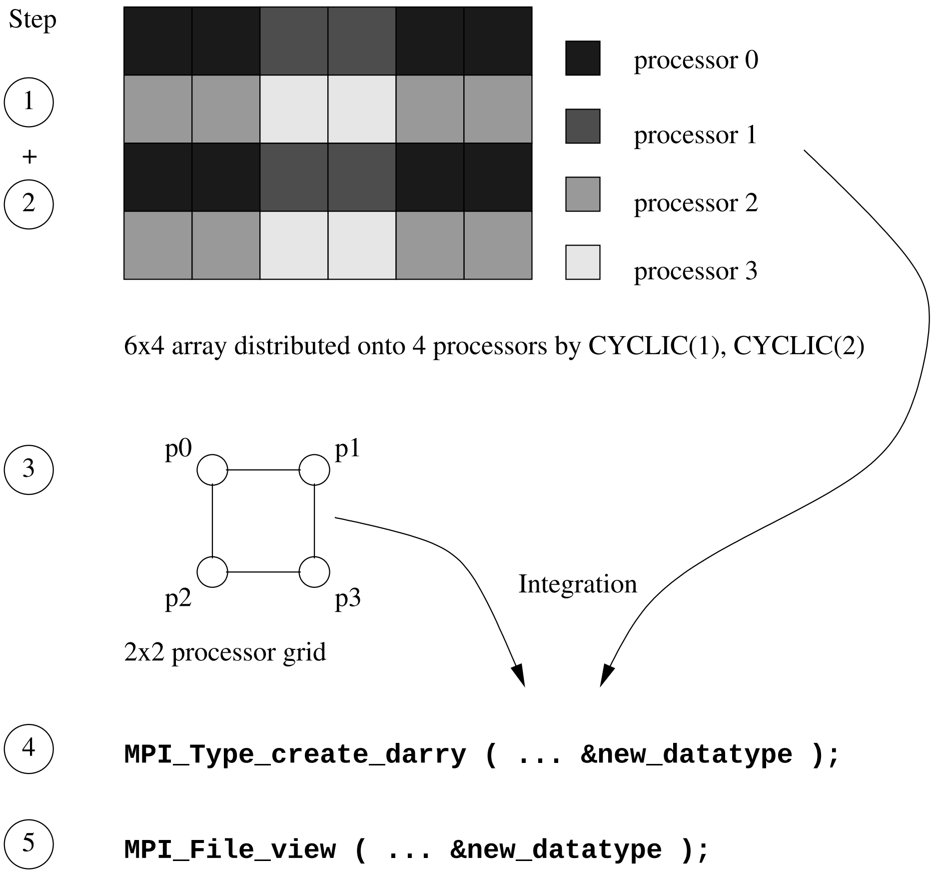

Figure 2. Formalized description of an MPI-IO program.

During the analysis and the projection phase of dimension 0, the results can be used directly. In other words, the array consists of 6 elements whereas 4 elements are to be accessed by processor 0. However, there is a stride of 2 elements between the second and the third basic block (see Fig. 3, Dim 0).

According to the next projection step, the access pattern of dimension 1 consists of 2 elements with a stride of one in between them (see Fig. 3, Dim 1). However, since the previous dimension (dimension 0) has to be considered as well, each basic block does not consists of only one data element but each basic block is made up by the access pattern defined by dimension 0. Even the stride between the two basic blocks is defined by the extent of the access pattern of dimension 0. Thus, although the number of elements of dimension 1 is merely 2 (two basic blocks), the number of elements actually accessed is 8 since each basic block consists of 4 elements (see Fig. 3, Result)

Finally, in the *Mapping phase*, the information yielded from the Integration phase must be mapped to the ViPIOS' internal data structure. We will not go into detail but merely describe the basic approach by means of a simple example. Again, a graphical analysis (see Fig. 4) shall motivate our approach.

According to Fig. 4 (part a) the access pattern of a single data block can be implemented by a simple MPI data type, e.g. MPI_INT. A contiguous access pattern (Fig. 4, part b) is then described by the derived data type MPI_Type_contiguous. A simple case access pattern (Fig. 4 part c) is implemented by means of MPI_Type_vector. Any other more complex access pattern can be realized by means of MPI_Type_struct, which allows defining data blocks of variable lengths. The access pattern depicted in Fig. 4, (part d) can be regarded as a typical complex case, since the workloads of processor 0 and processor 1 are different. However, internally the complex case for processor 1 is interpreted as a simple case (2 blocks of 2 elements) and a further simple case, namely one additional element. These two *simple cases* are combined by means of MPI_Type_struct to be treated as a complex case.

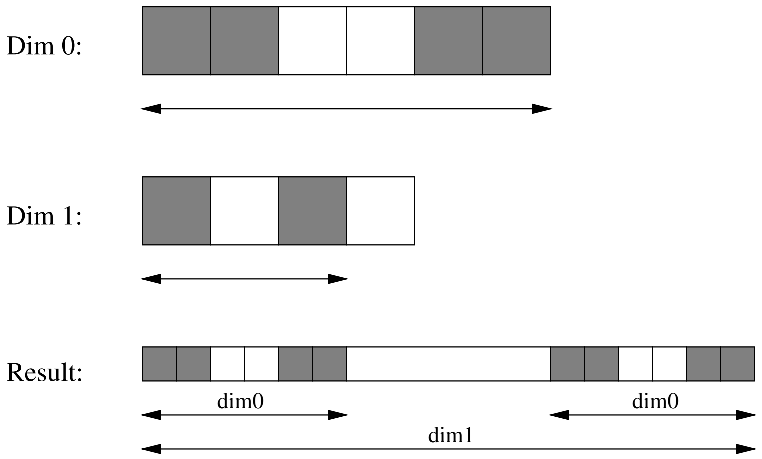

Figure 3. ViMPIOS distribution mapping.

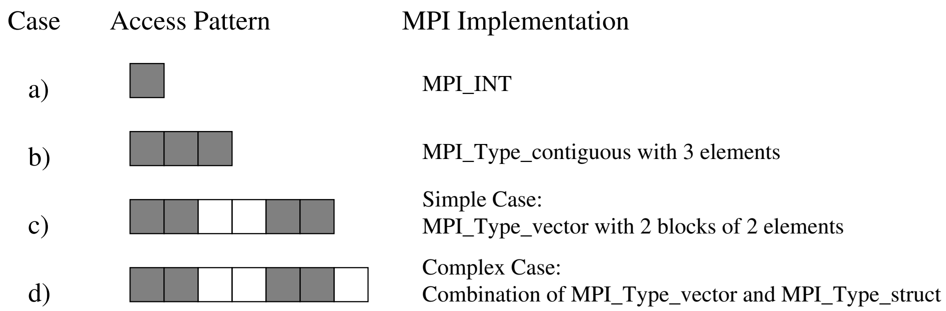

Figure 4. Mapping mechanism onto MPI data types.

5 ViPIOS Access Descriptor

On analyzing the access pattern and expressing it in terms of MPI derived data types, we can now map this information to the ViPIOS internal Access Descriptor. In particular, each value of the derived data type, which can be any complex combination of simple data types and derived data types, must be stored into the ViPIOS Access Descriptor. By means of defining offsets and strides, the access pattern of each processor as well as complex data types, which are not directly supported by MPI-IO derived data types, can be specified. Thus, the ViPIOS Access Descriptor can be used for mapping all MPI-IO derived data types and even more complex nested access patterns.

6 Analysis and Justification

A justification of our approach can easily be derived by a quite simple MPI-IO application. To show the advantage of our approach we used the program ”simple.c”, which is part of the test suite of the ROMIO package. It writes to separate files and reads the data back afterwards. Each MPI-IO process creates its own file which is distinguished by appending the process rank to the file name.

Our test environment consists of two available cluster systems. One cluster consists of Sun workstations connected by a conventional 10 Mbit Ethernet, the other is a Linux cluster with a 100 Mbit Fast-Ethernet interconnect.

First, we executed the MPI-IO program on the Sun cluster with 4 processors creating files of 4 MB size each. In the default installation of ROMIO the program creates its files locally in the ROMIO installation directory which is in a cluster environment by default linked via NFS to all cooperating processors. Thus, all 4 files are created on the same disk which leads to a sequential disk access scheme. The same program executed in the ViMPIOS environment creates automatically, totally transparently to the programmer, the files on the local disks of the respective processors. This is a result of the Fragmenter, the AI-based automatic data distribution module of the ViPIOS layer. The difference is about a factor of 5 in the execution times. The performance values are given in seconds in Tab. 1.

	ROMIO (default)	ROMIO (adapted)	ViMPIOS
Sun Cluster	27	5	5.5
Linux Cluster	15.1	2.5	2.9

Table 1. Performance comparison of ROMIO to ViMPIOS (time in sec.)

This can be corrected in ROMIO by the programmer choosing the respective local file system by changing the program code (which was not necessary in ViMPIOS). The respective results are comparative, even slightly better, to the ViMPIOS implementation[1].

Using the program on the Linux cluster delivers in the ViMPIOS environment a similar performance behaviour, but the ROMIO implementation need additional code adaptation to the MPI-IO program to direct the file accesses to the local files again. The difference in the absolute values between the Sun and Linux cluster is a result of the slower network interconnect.

Thus, it is easy to show that true portability (program migration without performance tradeoff) could only be reached by ViMPIOS. The ROMIO implementation needs additional programmers efforts.

7 Conclusion and Future Work

We presented ViMPIOS, a novel MPI-IO implementation based on ViPIOS, which adapts the problem specific data layout transparently to the user guaranteeing performance in changing I/O environments. Specifically, we described an algorithm to map MPI-IO data structures to corresponding ViPIOS structures. At the moment extensive performance tests are on the way, whereas the first results are very promising.

ViPIOS provides a wide spectrum of application interfaces, which spans from an I/O library to an independent data server system. In the near future the work will progress by the integration of an advanced buffer management module, the realization of a parallel file system implementation, and a Java interface.

References

[1] B.M. Chapman et al., *Vienna FORTRAN compilation system*, User's Guide, Vienna, 1994.

[2] Samuel A. Fineberg, Parkson Wong, Bill Nitzberg, and Chris Kuszmaul, *PMPIO— a portable implementation of MPI-IO*, Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation, IEEE Computer Society Press, October 1996, pp. 188–195.

[1]The slight performance advantage of ROMIO can be explained by the prototype implementation of ViMPIOS

[3] High Performance FORTRAN Forum, *High performance fortran language specification, version 1.1*, 1994.

[4] Message-Passing Interface Forum, *MPI-2.0: Extensions to the message-passing interface*, ch. 9, MPI Forum, June 1997.

[5] Erich Schikuta, Thomas Fuerle, and Helmut Wanek, *Vipios: The vienna parallel input/output system*, Euro-Par'98 (Southampton, England), Springer-Verlag, September 1998.

[6] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett, *Server-directed collective I/O in Panda*, Proceedings of Supercomputing '95 (San Diego, CA), IEEE Computer Society Press, December 1995.

[7] Rajeev Thakur, Ewing Lusk, and William Gropp, *Users guide for ROMIO: A high-performance, portable MPI-IO implementation*, Tech. Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, October 1997.

[8] Helmut Wanek and Erich Schikuta, *A blackboard approach for the automatic optimization of parallel I/O operations*, PaCT'99 (St. Petersburg, Russia), Springer-Verlag, September 1999.

[9] Richard W. Watson and Robert A. Coyne, *The parallel I/O architecture of the high-performance storage system (HPSS)*, Proceedings of the Fourteenth IEEE Symposium on Mass Storage Systems, IEEE Computer Society Press, September 1995, pp. 27–44.