ViMPIOS - A Portable, Client-Server Based Implementation of MPI-IO on ViPIOS

Kurt Stockinger

December 10, 1998

To my $\frac{twin}{2}$ or a slightly modified copy of the prototype...

Contents

1 Introduction **9**

2 MPI **11**
2.1 Introduction to MPI 11
2.2 The Basics of MPI 11
2.3 Linking and Running Programs 14
2.4 Blocking vs. Non-blocking communication 15
2.5 Derived Datatypes 17
2.6 Collective Communication 22
2.7 Communicators 23

3 MPI-IO **25**
3.1 Introduction to MPI-IO 25
3.2 First Steps With MPI-IO 26
3.2.1 Some Definitions of MPI-IO 28
3.3 MPI-IO routines 30
3.3.1 File Manipulation 32
3.3.2 File Info 36
3.3.3 File Views 36
3.3.4 Data Access 39
3.3.5 Data Access With Explicit Offset (_AT) 40
3.3.6 Data Access With Individual File Pointers 42
3.3.7 Split Collective Routines 45
3.3.8 Data Access With Shared File Pointers 46
3.3.9 Consistency and Semantics 47

4 ViPIOS Interface **50**
4.1 The ViPIOS Approach 50

4.2 System Architecture . . . 51
4.2.1 ViPIOS server . . . 52
4.2.2 System Modes . . . 53
4.3 Data Abstraction in ViPIOS . . . 55
4.4 ViPIOS Interface . . . 56
4.4.1 Connection and Disconnecting . . . 56
4.4.2 File Manipulation . . . 57
4.4.3 Data Access . . . 58
4.5 How to use the ViPIOS . . . 65
4.5.1 Quick Start . . . 65
4.5.2 An Example Program . . . 65
4.5.3 Strided Data Access . . . 67

5 ViMPIOS: Implementation of MPI-IO 71
5.1 Introduction to ViMPIOS . . . 71
5.2 Initializing and Finalizing MPI-IO . . . 72
5.3 File Management . . . 73
5.4 File Manipulation . . . 78
5.5 File Views . . . 79
5.5.1 The Mapping Function *get_view_pattern* . . . 82
5.6 Data Access Routines . . . 94
5.6.1 Read and Write . . . 94
5.6.2 Scatter Functionality . . . 99
5.6.3 Further Data Access Routines . . . 101
5.7 File interoperability and Consistency semantics . . . 107
5.8 Advanced Derived Datatypes . . . 107
5.8.1 Subarray Datatype Constructor . . . 108
5.8.2 Distributed Array Datatype Constructor . . . 110
5.8.3 Implementation of the Subarray Datatype Constructor 116
5.8.4 Implementation of the Distributed Array Datatype Constructor . . . 118
5.9 Fortran-Interface . . . 129

6 Regression Suite 137
6.1 Program Testmpio . . . 137
6.1.1 test_manycomms . . . 138
6.1.2 test_openmodes . . . 138
6.1.3 test_manyopens . . . 139

6.1.4 test_openclose 139
6.1.5 test_readwrite 139
6.1.6 test_rdwr 139
6.1.7 test_filecontrol 140
6.1.8 test_localpointer 140
6.1.9 test_collective 140
6.1.10 test_nb_rdwr 141
6.1.11 test_nb_localpointer 141

7 Test Results 142
7.1 The Performance of ViPIOS 142
7.1.1 Dedicated I/O Nodes 143
7.1.2 Non Dedicated I/O Nodes 146
7.2 Performance Comparison With Existing I/O Systems 148
7.2.1 ViPIOS compared to UNIX file I/O and MPI 148
7.2.2 ViPIOS Compared to ROMIO 149
7.3 Further Tests on ViPIOS and ViMPIOS 149
7.3.1 Scalability Results With Larger Files 150
7.3.2 ROMIO vs. ViMPIOS 151

8 Conclusion 153

List of Figures

2.1 Datatype constructor MPI_TYPE_VECTOR 19
2.2 Datatype constructor MPI_TYPE_INDEXED 20
2.3 Datatype constructor MPI_TYPE_STRUCT 22

3.1 File view 30
3.2 File view of 4 processes 31
3.3 Two different file views 31
3.4 Usage of displacements 39

4.1 Disk access de-coupling 51
4.2 ViPIOS system architecture 51
4.3 "Buddy" and "Foe" Servers 52
4.4 ViPIOS server architecture 52
4.5 ViPIOS system modes 54
4.6 ViPIOS data abstraction 54
4.7 File view with vectors 68
4.8 Two different file views with vectors 69
4.9 Level 1 70
4.10 Level 2 70

5.1 MPI_Type_hvector 87
5.2 MPI_Type_indexed 87
5.3 MPI_Type_struct 90
5.4 File view 103
5.5 Subarray 108
5.6 BLOCK(4)-distribution 111
5.7 CYCLIC(1)-distribution 111
5.8 BLOCK(4),CYCLIC(2)-distribution 112
5.9 BLOCK(4),BLOCK(2)-distribution with irregular patterns . . 112

5.10 Processor grid 114
5.11 Vector 117
5.12 4x6-array with BLOCK(2),BLOCK(3)-distribution 121
5.13 Vector distribution pattern 121
5.14 Regular and irregular distribution patterns 124
5.15 Distribation pattern of dimension 1 126
5.16 Distrubution pattern of both dimensions 127
5.17 Pattern of an irregular distribution 128
5.18 Irregular distribution pattern of one block 128

7.1 Topologies with different number of servers 143
7.2 Servers and clients reside on the same node 143
7.3 Servers and clients reside on different nodes 144
7.4 Program code 144
7.5 Scalability results 145
7.6 Non dedicated I/O nodes 147

List of Tables

7.1 Scalability results for dedicated I/O nodes 145
7.2 Constant workload 147
7.3 Increasing workload 148
7.4 Scalability results II for dedicated I/O nodes 151
7.5 Performance of ROMIO vs ViMPIOS 152

Acknowledgement

The first thank you goes to my advisor Prof. Erich Schikuta who introduced me to the research project at the Department of Data Engineering. After studying the theoretical background of solutions to the I/O problem from different universities, I got the possibility to take part in a project which implements a solution of its own. He always was a motivating advisor who cared for the right team spirit in the project. I also want to say thank you to my collegues and friends from the ViPIOS Research Project Thomas Fuerle and Helmut Wanek who answered my questions concerning the ViPIOS very patiently.

Besides the ViPIOS team I also owe a thank you to Prof. Gerald Quirchmayr who organized my year of study in London, UK, where I gained my first experience with parallel programming during a half unit project.

Chapter 1

Introduction

"There is great force hidden in a sweet command."

Over the last couple of years the computational performance of multiprocessors has increased quite fast. However, one of biggest problems of parallel computing is the so-called I/O bottleneck. Thus, limited I/O capacities can limit the performance of the whole system. In that report we present one approach of dealing with that problem, namely MPI-IO [9]. In short, MPI-IO is an extension to the MPI - Message Passing Interface that can be regarded as the de facto standard for parallel computing - and specifies a standard for parallel file I/O within the MPI.

The aim of that report is to describe in detail how this standard is implemented on the ViPIOS, which acts as the underlying parallel file system. In the first part we present a brief introduction to MPI with special emphasis on derived datatypes. These are user-defined datatypes that can be any combination of simple MPI datatypes and allow accessing sections of arrays which need not be contiguous in memory. Furthermore, we will explain terms like blocking and non-blocking calls. Then we will discuss the MPI-IO interface and give comprehensive information on all routines. For instance, we will define how the data access is performed by parallel application programs using MPI. By means of examples we will motivate our approach.

A further chapter deals with the description of the ViPIOS itself and its routines which are the basis for the implementation of MPI-IO. However, the main part of that report is dedicated to the description of the implementation process. In other words, we will analyze how the MPI-IO routines are actually programmed and how the internal structure of the ViMPIOS looks like. In particular we will discuss how the information of several files is handled. Since every strided data access of MPI-IO is based on derived datatypes, for example every tenth element of a file should be accessed, we will explain the mapping mechanism of MPI derived datatypes to the ViPIOS access descriptor so that an MPI call from the application program can be interpreted by the ViPIOS which in turn reads or writes the particular data. Thus, it is possible to access data in parallel, which is distributed over several processors.

Besides the C-interface we also focus our attention to the implementation of a corresponding FORTRAN-interface which maps the information from a FORTRAN application program to the C interface.

We will resume our discussion with a comprehensive test phase. Thus, application programs from other currently available MPI-IO implementations such as ROMIO [10] are used for two purposes. On the one hand, they serve as the basis for verifying the correctness of the ViMPIOS implementation, on the other hand, they are used to compare the performance of the two systems.

The last part of this thesis is dedicated to a performance study of the ViPIOS system. In particular, we will analyse the behaviour of parallel computation and compare these results yielded from the ViPIOS with ROMIO and the UNIX file system. The aim of this chapter is to give insights into the scalability of the ViPIOS. What is more we will demonstrate by means of some statisical data that the ViPIOS actualy does scale quite well.

We will conclude this thesis by a comparison of the performance of ROMIO with ViMPIOS. At the moment ROMIO outperforms ViMPIOS by a factor of about 2.5 for files larger than about 16MB. However, this can simply be explained by the fact that ROMIO uses the 2-Phase-Method [10] for internal I/O-optimization whereas ViPIOS does not provide any optimization algorithm so far. This will be a further step in the development of the overall system.

Chapter 2

MPI

2.1 Introduction to MPI

In this section we will discuss the most important features of the Message Passing Interface (MPI) [8]. Rather than describing every function in detail we will focus our attention to the basics of MPI which are vital to understand MPI-IO, i.e. the input/output part of the message passing interface. Thus, the overall purpose of this chapter is to define special MPI terms and explain them by means of the corresponding routines coupled with some examples.

The Message Passing Interface is the de facto standard for parallel programs based on the message passing approach. It was developed by the Message Passing Interface Forum (MPIF) with participation from over 40 organizations. MPI is not a parallel programming language on its own but a library that can be linked to a C or FORTRAN program. Applications can either run on distributed-multiprocessors, networks of workstations, or combinations of these. Furthermore, the interface is suitable for MIMD programs as well as for those written in the more restricted SPMD style. A comprehensive overview of parallel I/O terms can be found in [13].

2.2 The Basics of MPI

The main goal of the standard is to allow the communication of processes whereas the easiest way of interprocess communication is the point-to-point communication where two processes exchange information by the basic operations SEND and RECEIVE. According to [6] the six basic functions of MPI are as follows:

- MPI_INIT: initiate an MPI computation
- MPI_FINALIZE: terminate a computation
- MPI_COMM_SIZE: determine number of processes
- MPI_COMM_RANK: determine current process' identifier
- MPI_SEND: send a message
- MPI_RECV: receive a message

Every program in MPI must be initialized by *MPI_Init* and terminated by *MPI_Finalize*. Thus, no other MPI function can be called before *MPI_Init* or after *MPI_Finalize*. The syntax of the two functions is:

int MPI_Init (int *argc, char * argv)**
int MPI_Finalize (void)

By means of *MPI_Comm_rank* the process' identifier can be evaluated. Process numbers start with 0 and have consecutive integer values. In order to find out how many processes are currently running, *MPI_Comm_size* is called.

int MPI_Comm_size (MPI_Comm comm, int *size)

IN	comm	communicator
OUT	size	number of processes in the group of comm

int MPI_Comm_rank (MPI_comm, int *rank)

IN	comm	communicator
OUT	rank	rank of the calling process in group of comm

In both instructions the argument *comm* specifies a so-called communicator which is used to define a particular group of any number of processes. Suppose 8 processes are currently active and we wish to separate them into two groups, namely *group1* should contain processes with the identifiers from 0 to 3, whereas *group2* consists of the rest of the processes. Thus, we could use a communicator *group1* that refers to the first group and a communicator *group2* that refers to the second group is MPI_COMM_WORLD. This MPI predefined communicator includes all processes currently active.

On establishing a communication the next step is to explain how information is exchanged by *MPI_Send* and *MPI_Recv*. Both instructions execute a blocking message passing rather than a non-blocking one. In a blocking approach a send command waits as long as a matching receive command is called by another process before the actual data transfer takes place.

int MPI_Send (void* buf, int count, MPI_Datatype, int destination, int tag, MPI_Comm comm)

IN	buf	initial address of send buffer
IN	count	number of elements in the send buffer
IN	datatype	datatype of each send buffer element
IN	dest	rank of destination
IN	tag	message tag
IN	comm	communicator

int MPI_Recv (void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm, MPI_Status *status)

OUT	buf	initial address of receive buffer
IN	count	number of elements in the receive buffer
IN	datatype	datatype of each receive buffer element
IN	source	rank of source
IN	tag	message tag
IN	comm	communicator
OUT	status	status object

The first three arguments of both instructions are referred to as the message data, the rest is called message envelope. In particular *buf* specifies the initial address of the buffer to be sent. *Count* holds the number of elements in the send buffer, which are defined by the datatype *MPI_Datatype* (e.g. MPI_INT, MPI_FLOAT). The parameter *destination* states the identifier of the process that should receive the message. Similarly, the parameter *source* refers to the process that has sent the message. By means of *tag* a particular number can be related to a message in order to distinguish it from other ones. *Comm* refers to the communicator. Finally, the status information allows checking the source and the tag of an incoming message.

The following small program demonstrates how process 0 sends an array of 100 integer values to process 1:

```
#include "mpi.h"
int main (int argc,char **argv)
{
  int message[100], rank;
  MPI_Status status;

  /* MPI is initialized */
  MPI_Init(&argc,&argv);

  /* the rank of the current process is determined */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank==0)
    /*  process 0 sends message with tag 99 to process 1 */
    MPI_Send(message, 100, MPI_INT, 1, 99, MPI_COMM_WORLD);
  else
    /* process 1 receives message with tag 99 from process 0 */
    MPI_Recv(message, 100, MPI_INT, 0, 99, MPI_COMM_WORLD,
             &status);

  MPI_Finalize();
}
```

2.3 Linking and Running Programs

Compiling and linking of a program is done by

mpicc -o file_name file_name.c

On compiling and linking the program, an executable file is produced which can be executed by the following command:

mpirun -np number_of_processes file_name

-np denotes the number of process and has always be situated before the file name. For example:

mpirun -np 16 application.c

2.4 Blocking vs. Non-blocking communication

The send/receive commands we were discussing so far are so called blocking commands. In other words, the sending process waits until the receiving process has got the message. In contrast, nonblocking communication means that the sending processes do not wait until the operation is complete. Moreover, special functions, namely *MPI_Wait* and *MPI_Test*, are used to complete a nonblocking communication. Thus, better performance can be yielded for specific applications, since communication and computation can overlap.

Let us again take a look at the most important nonblocking commands before we resume with same examples:

int MPI_Isend (void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

IN	buf	initial address of send buffer
IN	count	number of elements in the send buffer
IN	datatype	datatype of each send buffer element
IN	dest	rank of destination
IN	tag	message tag
IN	comm	communicator
OUT	request	communication request

int MPI_Irecv (void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

OUT	buf	initial address of receive buffer
IN	count	number of elements in the receive buffer
IN	datatype	datatype of each receive buffer element
IN	source	rank of source
IN	tag	message tag

IN	comm	communicator
OUT	request	communication request

The syntax of those instructions is the same as for their blocking counterparts except of the last parameter *request*.

int MPI_Wait (MPI_Request *request, MPI_Status *status)

INOUT	request	request
OUT	status	status object

A call to that function returns when the operation identified by *request* is complete. In other words, it waits until a nonblocking send or receive with a matching parameter *request* is executed. *Status* gives information about the completed operation.

int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)

INOUT	request	communication request
OUT	flag	true if operation completed
OUT	status	status object

The function only returns *flag* = *true* if the operation identified by *request* is complete. Furthermore, the status object contains the information on the completed operation.

This example is a modification of the previous one. Rather than using blocking commands, we demonstrate the usage of non-blocking commands.

```
#include "mpi.h"
int main (int argc,char **argv)
{
  int message[100], rank;
  MPI_Status status;
  MPI_Request request;

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank==0)
```

```
  {
    MPI_ISend(message, 100, MPI_INT, 1, 99, MPI_COMM_WORLD,
              &request);
    /* any computation can be done here */
    MPI_Wait(request,status);
  }
  else
  {
     MPI_IRecv(message, 100, MPI_INT, 0, 99, MPI_COMM_WORLD,
              &request);
     /* any computation can be done here */
     MPI_Wait(request,status);
  }
  MPI_Finalize();
}
```

2.5 Derived Datatypes

Derived datatypes allow the user to define special datatypes that can be any combination of simple datatypes such as MPI_INT or MPI_FLOAT. In addition, sections of arrays that need not be contiguous in memory can be referred to as a special datatype. In that section we will give an overview of some important derived datatypes.

In order to define a derived datatype, three sets of functions are required, namely a constructor function to construct a derived datatype, e.g. *MPI_Type_vector*, a commit function *MPI_Type_commit* for applying the new datatype and finally the function *MPI_Type_free* that should be applied after the usage of the datatype.

The simplest derived datatype is *MPI_Type_contiguous*:

int MPI_Type_contiguous (int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

IN	count	replication count
IN	oldtype	old datatype
OUT	newtype	new datatype

This datatype allows defining a contiguous datatype which consists of *count* elements of a special datatype *oldtype*. The new datatype can be used for further purposes.

By means of an example we want to describe the usage of this datatype. Assume that process 0 wants to send an array of 25 integer elements to process 1. In order to use a derived datatype following steps are necessary:

```
MPI_Datatype array1;

/* datatype which specifies 25 integer values to contiguous
   locations is created */
MPI_Type_contiguous(25,MPI_INT,&array1);
MPI_Type_commit(&array1);

/* process 0 sends data to processes 1 */
MPI_Send(message, 1, array1, 1, 99, MPI_COMM_WORLD);
MPI_Type_free(&array1);
```

In our small example we only printed the code for process 0. Taking a look at the third parameter of the send command we notice that a derived datatype is specifed rather than a simple datatype like MPI_INT. The syntax for the commands which handle the commition and freeing of the derived datatype is given here:

int MPI_Type_commit (MPI_Datatype *datatype)
INOUT datatype   datatype that is commited

int MPI_Type_free (MPI_Datatype *datatype)
INOUT datatype   datatype that is freed

Now assume that process 0 wants to send sections of the array rather than all 25 integer elements. In particular, the first and the last 10 elements shall be sent which means that 2 blocks of data shall be sent whereas 5 elements shall be skipped. A more general derived datatype is *MPI_Type_vector*:

int MPI_Type_vector (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
IN   count   number of blocks

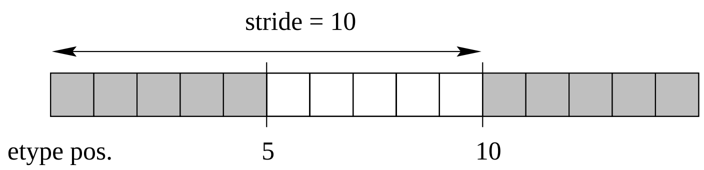

Figure 2.1: Datatype constructor MPI_TYPE_VECTOR

IN	blocklength	number of elements in each block
IN	stride	number of elements between start of each block
IN	oldtype	old datatype
OUT	newtype	new datatype

Count holds the number of blocks, *blocklength* specifies the number of elements in each block, and *stride* defines the number of elements between the start of each block whereas stride is a multiple of oldtype. The datatype for our example is as follows:

```
MPI_Type_vector(2,5,10);
```

The shape is depicted in Figure 2.1.

A further generalization of the previous datatype is *MPI_Type_hvector*. The difference to *MPI_Type_vector* is the parameter *stride* which is not given in elements but in bytes.

int MPI_Type_hvector (int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

IN	count	number of blocks
IN	blocklength	number of elements in each block
IN	stride	number of elements between start of each block
IN	oldtype	old datatype
OUT	newtype	new datatype

The next datatype *MPI_Type_indexed* allows specifying blocks of different lengths starting at different displacements. Before we present the function we want to give an example of such a particular case.

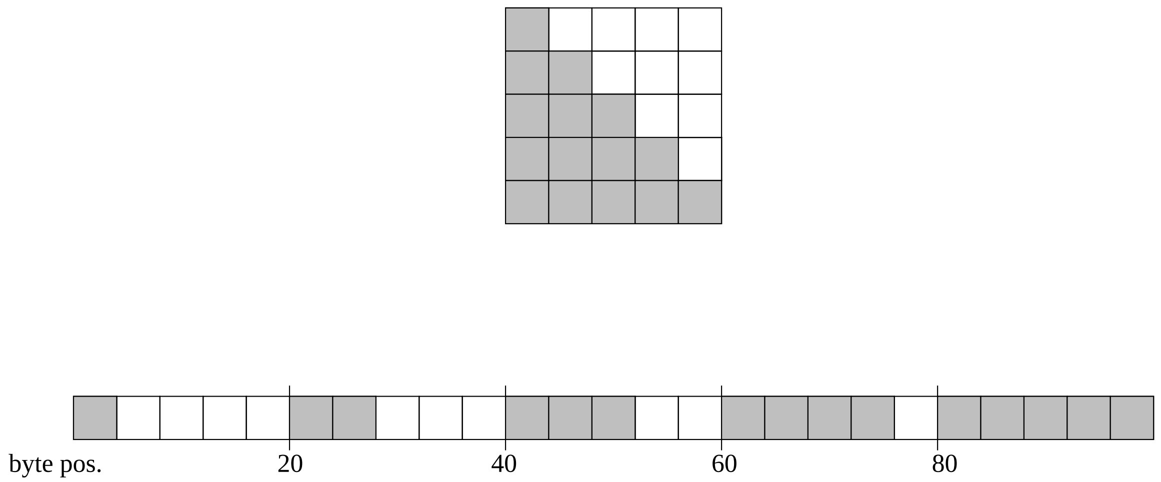

Figure 2.2: Datatype constructor MPI_TYPE_INDEXED

Assume a 5x5 array of integer values. Further assume that we want to send the lower triangle of that matrix. Thus, the first block to be sent consists of 1 element with the displacement 0. The second block consists of 2 elements with the displacement 6. The third block comprises 3 elements with the displacement 11 etc. The matrix and the corresponding linear file are depicted in Figure 2.2.

int MPI_Type_indexed (int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)

IN	count	number of blocks
IN	array_of_blocklengths	number of elements per block
IN	array_of_displacements	displacement for each block, in multiples of oldtype
IN	oldtype	old datatype
OUT	newtype	new datatype

```
MPI_Datatype    indexed;
int             a_blocklen[10],
a_disp[10];

for (i=0; i<5; i++)
{
```

```
  a_blocklen[i]=i+1;
  a_disp[i]=i*5;
}

MPI_Type_indexed(4, a_blocklen, a_disp, MPI_INT, &indexed);
MPI_Type_commit(&indexed);
```

Similar to *MPI_Type_hvector* there is also a corresponding *MPI_Type_hindexed*. This datatype is identical to the previous except that the displacements are given in bytes and not in multiples of *oldtype*.

int MPI_Type_hindexed (int count, int *array_of_blocklenghts, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)

The most general derived datatype is *MPI_Type_struct*. The difference to *MPI_Type_indexed* is that each datablock can consist of a different datatype. Thus, it is possible to send an integer, together with a character or double value in one message.

int MPI_Type_struct (int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_oldtypes, MPI_Datatype *newtype)

IN	count	number of blocks
IN	array_of_blocklengths	number of elements per block
IN	array_of_displacements	displacement for each block in bytes
IN	array_of_oldtypes	old datatype
OUT	newtype	new datatype

Let us again take a look at an example:

```
MPI_Datatype    s_types[3]={MPI_INT,MPI_DOUBLE,MPI_CHAR},
dd_struct;
int             s_blocklen[3]={3,2,16};
MPI_Aint        s_disp[3];

/* set displacements to next free space */
s_disp[0]=0;
```

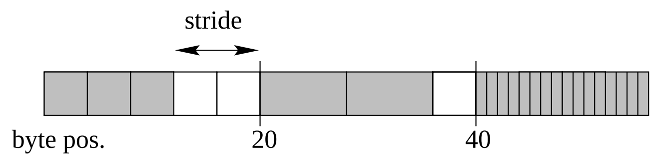

Figure 2.3: Datatype constructor MPI_TYPE_STRUCT

```
s_disp[1]=20;
s_disp[2]=40;

MPI_Type_struct(3, s_blocklen, s_disp, s_types, &dd_struct);
MPI_Type_commit(&dd_struct);
```

The shape of that derived datatype can be seen in Figure 2.3. Here the derived datatype consists of 3 non-contiguous blocks of different datatypes, namely MPI_INT, MPI_DOUBLE and MPI_CHAR.

2.6 Collective Communication

Unlike the point-to-point communication we stated in the previous chapters, collective communication means that a group of processes can take part in the communication process. In other words, not only one process can send a message to another processes but to several processes with just one command. In contrast, it is also possible for one process to receive messages from all other processes specified in one group. Again, only one receive command is necessary rather than several ones for each message.

Let us take a look at one function for collective communication in order to clarify the advantage of a collective communication over a point-to-point communication:

int MPI_Bcast (void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

INOUT	buffer	starting address of buffer
IN	count	number of entries in buffer
IN	datatype	datatype of buffer
IN	root	rank of process to broadcast the message

IN comm communicator

This routine broadcasts a message from the process identified by the rank *root* to all processes in the group. What is more, a message is also sent to itself.

Let us first analyze how this would be done with the conventional point-to-point communication and compare this to the collective version. Assume that our group consists of 8 processes. The message to be sent should be an array of 1000 double values.

```
MPI_Comm comm;
double  double_array[1000];
...
if (rank==0)
{
  for (i=0; i<8; i++)
    MPI_Send(double_array, 1000, MPI_DOUBLE, i, 99, comm);
}
else
  MPI_Recv(double_array, 1000, MPI_DOUBLE, 0, 99, comm, &status);
```

Using collective routines the send-receive process is much shorter since it is not necessary to write separate lines of code for different processes:

```
MPI_Bcast(double_array, 1000, MPI_DOUBLE, 0, comm);
```

2.7 Communicators

Communicators are a way of managing the communication among processes in a better way such that several processes can be grouped together and regarded as one homogenous entity. In short, communicators are divided into to kinds:

- Intra-Communicators
- Inter-Communicators

The first term refers to the communication within a single group of processes whereas the second term refers to the point-to-point communication between two group of processes. For a comprehensive survey of all MPI commands we refer the reader to [8].

Chapter 3

MPI-IO

3.1 Introduction to MPI-IO

MPI-IO is a high level interface developed by the MPI-Commettee [8]. The goal was to create a widely used standard for describing parallel I/O operations within an MPI message passing application. The initial idea was that I/O could be modeled as message passing, namely writing to a file can be regarded as sending a message and reading from a file is equivalent to receiving a message. Thus, MPI-IO is an extension of the MPI 1.1 standard , which did not support parallel file I/O so far for the following reasons [6]:

- not all parallel machines support the same parallel or concurrent file system interface
- the traditional Unix file system interface is ill suited to parallel computing since multiple processes do not share files at once

On giving a short introduction to the I/O problem let us now analyze the most important features of MPI-IO:

- using derived MPI datatypes yields strided access to the memory and the file
- non-blocking functions improve the I/O performance by overlapping I/O with computation
- collective operations may optimize global data access

- using two types of file pointers, namely individual and shared file pointers, such that exact offsets in the file need not be specified when data is read or written
- file hints allow specifying the layout of the a file, e.g. number of disks or I/O nodes which hold the information of a striped file
- filetype constructors can be used to specify array distribution
- error handling is supported

3.2 First Steps With MPI-IO

Referring to the introductory chapter we stated that I/O can be modeled as a message passing system. In other words, writing data to a file should be similar to sending a message. In contrast, reading from a file should be modeled as receiving a message. Although MPI-IO supports a large number of different functions for parallel I/O, many programs and applications only use six of them, which are summarized in the following paragraph [6]:

- MPI_INIT: MPI as well as MPI-IO are initialized
- MPI_FILE_OPEN: a file is opened
- MPI_FILE_READ: data is read from a particular location in a file
- MPI_FILE_WRITE: data is written to a particular location in a file
- MPI_FILE_CLOSE: a file is closed
- MPI_FINALIZE: MPI as well as MPI-IO are terminated

Strictly speaking only four so-called MPI-IO functions are used since MPI_INIT and MPI_FINALIZE are already supported by MPI-1.

We will now explain the use of these functions by means of two simple examples [6]. For introductory purpose we will not go into detail with describing the exact syntax of each function but only mention the most important parameters to focus our attention. We will dedicate a special chapter to the syntax of the functions at a later stage of this thesis.

In the first program each process creates its own individual file called *file* followed by an extension which reflects the identifier of the current process. A file can be opened individually by using the parameter MPI_COMM_SELF. Furthermore, data is written to the file, which is read back later on.

```
#include "mpi.h"
#include "mpio.h"

int main(argc,argv)

int argc;
char *argv[];
{
  int myid;

  MPI_Status status;
  MPI_File fh;

  char filename[12];

  buf= (int *)malloc(50*sizeof(int));
  for (i=0, i<50, i++)
    buf[i]=i;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  /* open file with filename "file.processid" */
  sprintf(filename,"%s.%d","ufs:file",myid);

  /* each process opens a separate file */
  MPI_File_open (MPI_COMM_SELF, filename, MPI_MODE_CREATE |
MPI_MODE_RDWR, MPI_INFO_NULL, &fh );

  /* read data from file */
  MPI_File_read(fh, buf, 50, MPI_INT, &status);
```

```
  /* perform computation */

  /* write data to file */
  MPI_File_write (fh, buf, 50, MPI_INT, &status);

  MPI_File_close(&fh);

  free(buf);
  MPI_Finalize();

}
```

In the second example each process accesses one common global file rather than its local one. That feature is yielded by the parameter MPI_COMM_-WORLD. We will not print the whole program code but only the part which differs from the previous example:

```
/* each process opens one common file */
MPI_File_open (MPI_COMM_WORLD, filename, MPI_MODE_CREATE |
MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
```

3.2.1 Some Definitions of MPI-IO

File: An MPI file is an ordered collection of typed data items which can be accessed in a random or sequential way. Furthermore, a communicator (MPI_COMM_SELF or MPI_COMM_WORLD we discussed in our introductory chapter) on the one hand specifies which group of processes can get access to the I/O operations, on the other hand, it determines whether the access to the file is independent or collective. Since independent I/O requests are executed individually by any of the processes within a communicator group, no coordination among the processes is required. In contrast, the latter case requires each process in a group associated with the communicator to participate in the collective access.

Displacement: A file displacement defines the beginning of a view (file access pattern) expressed as an absolute byte position relative to the beginning of the file. Furthermore, it can be used to skip head information of a file or to define further access patterns which start at different positions of the file.

Etype: An etype (elementary datatype) can be regarded as the unit of data access and positioning. In other words, it specifies the data layout in the file. An etype can be any MPI predefined or derived datatype.

Filetype: A filetype can either be a single etype or a derived MPI datatype of several etypes and describes a template for accessing a file partitioned among processes.

View: A view is described by a displacement, an etype, and a filetype and defines the current set of data visible and accessible from an open file as an ordered set of etypes. Thus, a view specifies the access pattern to a file.

Offset: An offset is a position in the file relative to the current view, expressed as a count of etypes.

File pointer: A file pointer is an implicit offset. On the one hand, MPI provides individual file pointers which are local to each process, on the other hand, file pointers for a group of processes - so-called shared file pointers - are supported.

File handle: A file handle can be regarded as a file reference which is created by *MPI_File_open* and freed by *MPI_File_close*.

By means of an example we will explain the idea of strided access and how this can be achieved by using different file views. Assume a file which holds an array of 24 integer values. Further assume that a process only wants to read or write every third value of the file. We therefore tile the file with a filetype which is a derived MPI datatype. Thus, the values at position (offset) 0,2,5,... can be accessed whereas all the other positions are so-called holes which cannot be accessed by the current process. The view is depicted in Figure 3.1.

In our next example assume that 3 processes access the file in a complementary way. In other words, the file is partitioned among four parallel processes where each process reads or writes at different locations in the file. In particular process 0 accesses the positions 0,3,6,..., process 1 accesses the positions

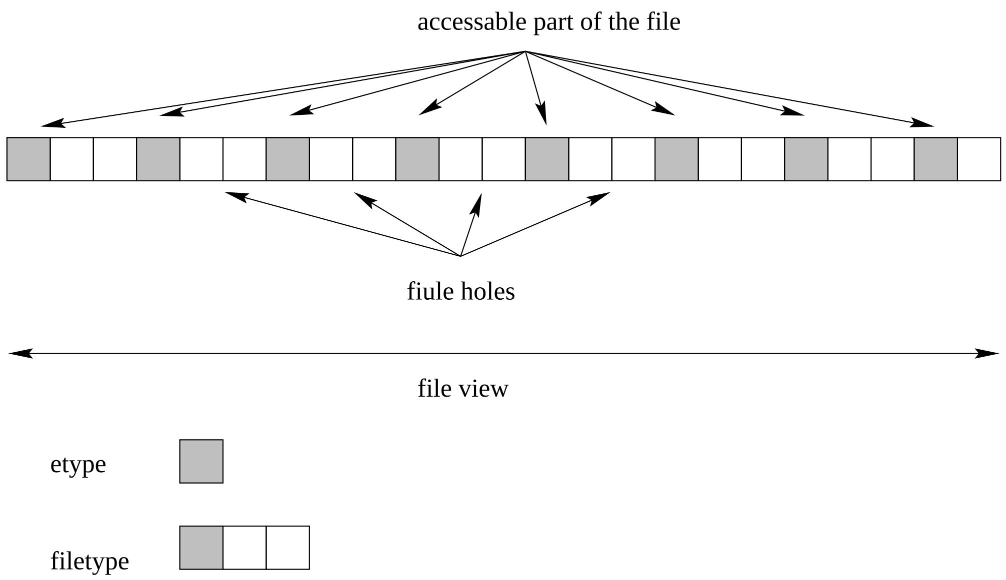

Figure 3.1: File view

1,4,7,..., and finally process 2 accesses the positions 2,5,8,... The access patterns of each process are depicted in Figure 3.2.

Another possibility is to access a file in two different patterns. Thus, we define two tilings (views). In particular, the first part of the file shall be accessed with a stride of 2 elements whereas the second part of the file shall be accessed with a stride of 3 elements. Figure 3.4 can make this example clearer:

If the file needs to be accessed in the second view, the displacement of the second pattern will skip over the entire first segment.

3.3 MPI-IO routines

On giving some definitions of the MPI-IO standard we will now describe the routines of that standard and give some examples to get a first idea of how MPI-IO works.

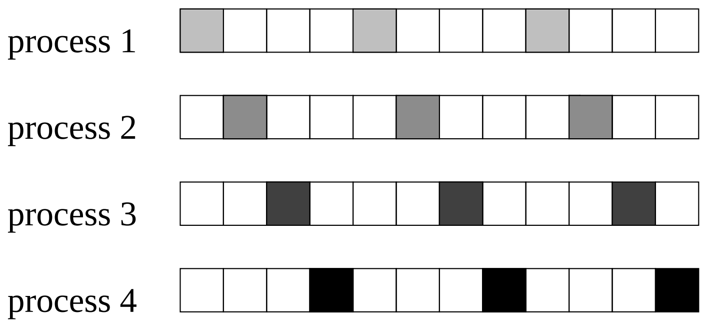

Figure 3.2: File view of 4 processes

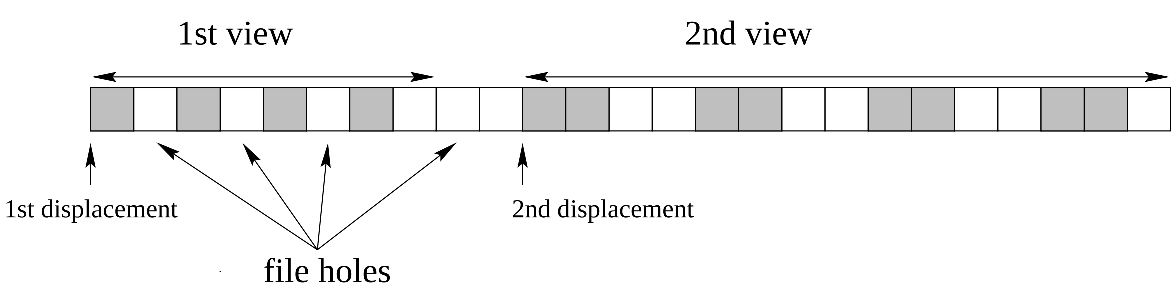

Figure 3.3: Two different file views

3.3.1 File Manipulation

int MPI_File_open (MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)

IN	comm	communicator
IN	filename	name of file to be opened
IN	amode	file access mode
IN	info	info object
OUT	fh	new file handle (handle)

Description:
Opens the file *filename* on all processes in the *comm* communicator group.

Features:

- collective routine (all processes must refer to the same file name and use the same access mode)
- a file can be opened independently by using the MPI_COMM_SELF communicator
- *comm* must be an intra-communicator rather than an inter-communicator
- *filename* is prefixed to indicate the underlying file system (e.g. "ufs:filename", where ufs stands for Unix file system)
- *amode* can have the following values which could also be combined together with the bit vector OR:
  - MPI_MODE_RDONLY: file is opened read only
  - MPI_MODE_RDWR: reading and writing
  - MPI_MODE_WRONLY: write only
  - MPI_MODE_CREATE: create the file if it does not exist
  - MPI_MODE_EXCL: error is returned if a file is created that already exists
  - MPI_MODE_DELETE_ON_CLOSE: delete file on close
  - MPI_MODE_UNIQUE_OPEN: file will not be concurrently opened elsewhere. Optimization is yielded by eliminating file locking overhead

  - MPI_MODE_SEQUENTIAL: file will only be accessed sequentially
  - MPI_MODE_APPEND: set initial position of all file pointers to end of file

- *info* provides information such as file access patterns and file system specifics. If no info is needed, MPI_INFO_NULL can be used.

Annotation:
Every time the file is accessed the file handle *fh* is used to refer to the file. Moreover, each open-function must at least contain either MPI_MODE_RDONLY, MPI_MODE_RDWR or MPI_MODE_WRONLY

Possible errors:

- MPI_MODE_CREATE or MPI_MODE_EXCL are specified together with MPI_MODE_RDONLY
- MPI_MODE_SEQUENTIAL is specified together with MPI_MODE_RDWR
- a file which is opened with MPI_MODE_UNIQUE is opened concurrently
- a file which is opened with MPI_MODE_SEQUENTIAL is accessed in a non-sequential way

int MPI_File_close (MPI_File *fh)
INOUT fh file handle

Description:
The file defined by *fh* is closed after synchronizing the file, i.e. previous writes to *fh* are transferred to the storage device. Furthermore, the content of the file handle *fh* is destroyed.

Features:

- collective routine
- the file is deleted if it was opened with MPI_FILE_DELETE_ON_CLOSE

int MPI_File_delete (char *filename, MPI_Info info)

IN	filename	name of the file
IN	info	file info

Description:
Deletes the file *filename*.

Possible errors:

- MPI_ERR_NO_SUCH_FILE: a file is attempted to delete which does not exist
- MPI_ERR_FILE_IN_USE, MPI_ERR_ACCESS: the file is still opened by any other process

int MPI_File_set_size (MPI_File fh, MPI_Offset size)

INOUT	fh	file handle
IN	size	size (in bytes) to truncate or expand file

Description:
Resizes the file defined by *fh*.

Features:

- collective routine (identical value for *size*)
- if *size* is smaller than the current file size, the file is truncated otherwise *size* becomes the new file size
- does not effect individual or shared file pointers

Possible errors:

- routine is called if file was opened with MPI_MODE_SEQUENTIAL
- non-blocking requests and split collective operations on *fh* are not completed

int MPI_File_preallocate (MPI_File fh, MPI_Offset size)

INOUT	fh	filehandle
IN	size	size to preallocate file size

Description:
Storage space is allocated for the first *size* bytes of a file.

Features:

- collective routine (identical value for *size*)
- if *size* is larger than the current file size, the file is increased to *size* otherwise the file size is unchanged

Possible errors:

- routine is called if file was opened with MPI_MODE_SEQUENTIAL

int MPI_File_get_size (MPI_File fh, MPI_Offset size)

IN	fh	file handle
OUT	size	size of the file in bytes

Description:
Returns the current size in bytes of the file defined by *fh*.

int MPI_File_get_group (MPI_File fh, MPI_Group *group)

IN	fh	file handle
OUT	group	group which opened the file

Description:
Returns a duplicate of the group of the communicator which opened the file defined by *fh*.

int MPI_File_get_amode (MPI_File fh, int *amode)

IN	fh	file handle
OUT	amode	file access mode used to open the file

Description:
Returns the access mode of the file defined by *fh*.

3.3.2 File Info

int MPI_File_set_info (MPI_File fh, MPI_Info info)

INOUT	fh	file handle
IN	info	info object

Description:
Sets new values for the hints of a file.

int MPI_File_get_info (MPI_File fh, MPI_Info *info_used)

IN	fh	file handle
OUT	info_used	new info object (handle)

Description:
Returns the hints of a file.

Some examples of file hints:

- Access_style: Specifies how a file is accessed whereas a combination of following strings is possible: read_once, write_once, read_mostly, write_mostly, sequential, reverse_sequential and random
- io_node_list: Specifies the list of I/O devices to store the file

3.3.3 File Views

Strided access to a file can be gained by using derived datatypes in combination with views. Recall that a view defines the current set of data visible and accessible from an open file as an ordered set of etypes.

Let us first present the interfaces before we give a simple example.

int MPI_File_set_view (MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char *datarep, MPI_Info info)

INOUT	fh	file handle
IN	disp	displacement
IN	etype	elementary datatype
IN	filetype	filetype
IN	datarep	data representation
IN	info	info object

Description:
Changes the process' view of the data in the file.

Features:

- collective routine: values for *etype* and *datarep* must be identical on all processes in the group
- *disp* defines the beginning of the view in bytes, it must have the special value MPI_DISPLACEMENT_CURRENT if MPI_MODE_SEQUENTIAL was specified
- *etype* defines the data access and positioning in the file. Thus, every seek which is performed on that file is done in units specified by *etype*. In addition every offset is expressed as a count of *etypes*. More information is given when the command *MPI_File_seek* is explained.
- *filetype* describes the distribution of the data
- *datarep* specifies the representation of the data in the file according to three categories:
  - native
  - internal
  - external32

Annotation:
In the *native* data representation the data is stored in a file exactly as it is in memory [9]. Since no type conversion is required for that mode, data precision and I/O performance are not lost. With the *internal* data representation type conversion is performed if necessary. Type conversion is always performed with the *external32*.

Non-blocking requests and split collective operations (see in a later chapter) on. *fh* must be completed

int MPI_File_get_view (MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char *datarep)

IN	fh	file handle
OUT	disp	displacement

OUT	etype	elementary datatype
OUT	filetype	filetype
OUT	datrep	datarepresentation

Description:
Returns the process' view of the data in the file.

Features:

- **Datarep** must be large enough to handle the data representation string. However, the upper limit is defined by MPI_MAX_DATAREP_STRING

Example: Assume a file which is an integer array consisting of 100 values starting from 0. Further assume that this file should be accessed in 10 blocks of size 2 with a stride of 10. The corresponding part of the program looks like follows:

```
MPI_File_open (MPI_COMM_WORLD, "ufs:file1", MPI_MODE_CREATE |
    MPI_MODE_RDWR, MPI_INFO_NULL, &fh );

MPI_Type_vector (10,2,10,MPI_INT,&newtype1);
MPI_Type_commit(&newtype1);
MPI_File_set_view (fh, 0, MPI_INT, newtype1, "native",
                                  MPI_INFO_NULL);
MPI_File_read(fh, buf, 10, MPI_INT, &status);
```

In that example the displacement is 0. This means that the view begins at position 0 of the file. The displacement can also be used to skip some header information or to specify a second view of a file which begins at a position other than 0 and can be accessed in a different pattern. See Figure 3.4.

Example: Again we assume our file which is an integer array consisting of 100 values. We now want to define a view for the second part of the file such that we can access every second value. The new derived datatype and the corresponding view look like follows:

```
MPI_Vector (25,1,2, &newtype2);
MPI_Commit (&newtype2);
MPI_File_set_view (fh, 200, MPI_INT, newtype2, "native",
                                  MPI_INFO_NULL);
```

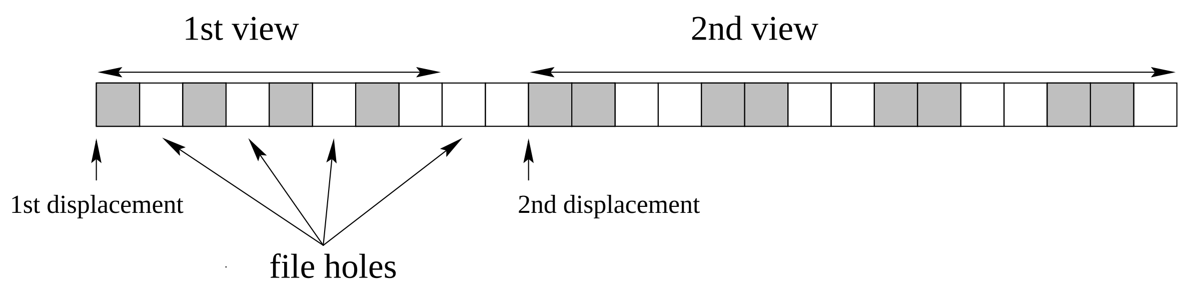

Figure 3.4: Usage of displacements

Since the displacement is given as an absolute offset in bytes from the beginning of the file, the value 200 sets the file view to the 50th element of the file. This is only true if the size of an integer type is 4 bytes.

3.3.4 Data Access

On analyzing different aspects of file manipulation we will now discuss the three fundamental aspects of data access which are as follows:

- positioning (explicit offset vs. implicit file pointer)
- synchronism (blocking vs. non-blocking and split collective)
- coordination (collective vs. non-collective)

Positioning: All MPI routines that use explicit offsets for accessing data contain *_at* in their name (e.g. *MPI_File_read_at*). Rather than manipulating individual file pointers or shared file pointers data is accessed at the position defined by an argument. The different positioning methods can be mixed in the same program without effecting each other. This means that changing a file by an explicit offset will not affect the individual file pointer since no file pointer is used or updated.

Synchronism: Besides blocking I/O routines MPI also supports non-blocking I/O routines which are named *MPI_File_ixxx*, where *i* refers to immediate. Similar to MPI, a separate request complete call (*MPIO_Wait, MPIO_Test*) is needed to complete the I/O request. However, this does not mean that the data is written to a "permanent" storage. The MPI-IO function that deals with that case is *MPI_File_sync* which we will discuss later on.

Coordination: The collective counterpart to the non-collective data access routine *MPI_File_xxx* is *MPI_File_xxx_all*. This means that a collective call is only completed when all other processes that participate in the collective call have completed their tasks. A restricted form of "non-blocking" operations for a collective data access is called *split collective*. Rather than using one routine like *MPI_File_xxx_all*, a pair of *MPI_File_xxx_begin* and *MPI_File_xxx_end*is used. Thus, a single collective operation is separated by two routines where the begin routine can be compared to a non-blocking data access like *MPI_File_write* and the end routine acts like a matching routine which completes the operation, e.g. *MPI_Wait* or *MPI_Test*. In addition, the counterparts to the *MPI_File_xxx_shared* routines are *MPI_File_xxx_ordered*.

Conventions of Data Access

Every file which is used by MPI-IO commands is referred to by its file handle *fh*. The parameters *buf*, *count*, and *datatype* specify how the data is stored in memory. Note that similar to a receive, *datatype* must not contain any overlapping regions. Finally, *status* returns the amount of data accessed by a particular process. In detail, the number of datatype entries and predefined elements can be retrieved by calling *MPI_get_count* and *MPI_get_elements*.

3.3.5 Data Access With Explicit Offset (_AT)

The routines listed in that section can only be used if MPI_MODE_SEQUENTIAL was not specified.

Blocking, non-collective:

int MPI_File_read_at (MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_status status)

IN	fh	file handle
IN	offset	file offset
OUT	buf	initial address of buffer
IN	count	number of elements in the buffer
IN	datatype	datatype of each buffer element
OUT	status	status object

Description:
Reads a file beginning at position *offset*. *Status* contains the amount of data accessed by that routine.

Annotation:
The offset for all explicit offset routines is given in units of *etype* rather than in bytes. Furthermore, *offset* expresses the position relative to the beginning of a file.

int MPI_File_write_at (MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_status status)

INOUT	fh	file handle
IN	offset	file offset
IN	buf	initial address of buffer
IN	count	number of elements in the buffer
IN	datatype	datatype of each buffer element
OUT	status	status object

Description:
Writes a file beginning at position *offset*. *Status* contains the amount of data accessed by that routine.

Collective Versions:

Since the semantic of the collective routines are the same as for their non-collective counterparts, only the synopsis of the routines are printed here:

int MPI_File_read_at_all (MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_status status)

int MPI_File_write_at_all (MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_status status)

Non-blocking Versions:

Note that non-blocking I/O calls only initiate I/O operations but do not wait for them to complete. Thus, separate request complete calls like MPIO_WAIT or MPIO_TEST are needed. The example in the next section will demonstrate the use of non-blocking functions.

int MPI_File_iread_at (MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Request request)

IN	fh	file handle
IN	offset	file offset
OUT	buf	initial address of buffer
IN	count	number of elements in the buffer
IN	datatype	datatype of each buffer element
OUT	request	request object

int MPI_File_iwrite_at (MPI_File fh, MPI_Offset offset, void *buf, int count MPI_Datatype datatype, MPI_Request request)

INOUT	fh	file handle
IN	offset	file offset
IN	buf	initial address of buffer
IN	count	number of elements in the buffer
IN	datatype	datatype of each buffer element
OUT	request	request object

3.3.6 Data Access With Individual File Pointers

One individual file pointer per process per file handle defines the offset of the data to be accessed. Since the semantics are the same as in the previous section we will not go into detail with describing the routines and refer to table.

int MPI_File_read (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status status)

int MPI_File_write (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status status)

int MPI_File_seek (MPI_File fh, MPI_Offset offset, int whence)

INOUT	fh	file handle
IN	offset	file offset
IN	whence	update mode (state)

Description:
Updates the individual file pointer according to whence whereas following features are possible:

- MPI_SEEK_SET: file pointer is set to offset
- MPI_SEEK_CUR: file pointer is set to current pointer position plus offset
- MPI_SEEK_END: file pointer is set to the end of the file plus offset

Annotation:
It is important to mention that the offset is not given in bytes but in units of *etype* defined by the view of the file.

int MPI_File_get_position (MPI_File fh, MPI_Offset *offset)

IN	fh	file handle
OUT	offset	offset of file pointer

Description:
Returns the current position of the individual file pointer in *etype* units relative to the current view.

int MPI_File_get_byte_offset (MPI_File fh, MPI_Offset offset, MPI_Offset *disp)

IN	fh	file handle
IN	offset	offset of filepointer
OUT	disp	absolute byte position of offset

Description:
Converts a view-relative offset which is given in *etype* units into an absolute byte position relative to the current view.

Besides blocking, non-collective routines which we presented here, non-blocking or collective routines are defined in the MPI-IO standard as well.

Example: In this example we want to demonstrate that in general each I/O operation leaves the file pointer pointing to the next data item after the last one that is accessed by the operation. In other words, file pointers are updated automatically. Again we use our file which is an array consisting of 100 values. We assume that the first ten values should be stored in the array *buf1[]* and the following 10 values in the array *buf2[]*. Note that after the first non-blocking read the file pointer is adjusted automatically to position 10 in the array. Furthermore, the file pointer is not updated by a routine with an explicit offset.

```
MPI_Status status1,status2;
MPIO_Request request1, request2;

MPI_File_open (MPI_COMM_WORLD, "ufs:file1", MPI_MODE_CREATE |
MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native",
MPI_INFO_NULL);


/* File pointer points to position 0 of the file */
MPI_File_iread(fh, buf1, 10, MPI_INT, &request1);

/* File pointer points to position 10 since 10 values are read
   by the previous routine */
MPI_File_iread(fh, buf2, 10, MPI_INT, &request2);

/* File pointer points to position 20 of the file since another
   10 values are read */
MPI_File_read_at(fh, 51, buf3, 10, MPI_INT, &request1);

/* File pointer still points to position 20 since previous read
   is a routine with explicit offset that does not update the
   file pointer */
MPI_File_read_(fh, buf4, 10, MPI_INT, &request1);

MPIO_Wait(&request1,&status1);
MPIO_Wait(&request2,&status2);
MPI_File_close(&fh);
```

On executing the program the arrays contain following values:

```
buf1: 0 1 2 3 4 5 6 7 8 9
```

```
buf2: 10 11 12 13 14 15 16 17 18 19
buf3: 50 51 52 53 54 555 56 57 58 59
buf4: 20 21 22 23 24 25 26 27 28 29
```

3.3.7 Split Collective Routines

We have already defined a split collective routine as a restricted form of "non-blocking collective" I/O operations. Before we present the interface routines let us first state the most important semantic rules according to the MPI-IO standard:

- Each file handle on any MPI process must not have more than one active split collective operation at any time
- Begin calls are collective and must be followed by a matching collective end call
- Regular collective operations like MPI_FILE_WRITE_ALL on one process do not match split collective operations on another process.
- Split collective routines must not be used concurrently with collective routines

```
MPI_File_write_all_begin(fh,...);
...
/* This collective routine is used concurrently with a split
   collective routine */
MPI_File_read_all(fh,...);
...
MPI_File_write_all_end(fh,...);
```

Again the semantics of these operations are the same as for the corresponding collective operations. We therefore only present one example of split collective routines:

int MPI_File_read_at_all_begin (MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype)

IN	fh	file handle
IN	offset	file offset

OUT	buf	initial address of buffer
IN	count	number of elements in the buffer
IN	datatype	datatype of each buffer element

int MPI_File_read_at_all_end (MPI_File fh, void *buf, MPI_status status)

IN	fh	file handle
OUT	buf	initial address of buffer
OUT	status	status object

3.3.8 Data Access With Shared File Pointers

The offset in the data access routine is described by exactly one shared file pointer per collective MPI_FILE_OPEN. Thus, the file pointer is shared among the processes in a particular communicator group. Again, the same semantics are used as in the previous sections with some exceptions:

- All processes must use the same view.
- Using collective shared file pointers (_ORDERED) guarantees a serialized order of multiple calls. In other words, the access to the file is determined by the rank of the processes in the group. In contrast, non-collective shared file pointers (_SHARED) yield a serialization ordering which is non-deterministic.

The functions listed below shall be regarded as a proposal since they are not supported by ViMPIOS yet.

Non-collective Routines:

int MPI_File_read_shared (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status status)

int MPI_File_write_shared (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status status)

int MPI_File_seek_shared (MPI_File fh, MPI_Offset offset, int whence)

int MPI_File_get_position_shared (fh, offset)

Collective Routines:

int MPI_File_read_ordered (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status status)

int MPI_File_write_ordered (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status status)

3.3.9 Consistency and Semantics

We can distinguish three different levels of consistency [9]:

- sequential consistency among all processes using a single file handle
- sequential consistency among all processes using file handle created from a single collective open with atomic mode enabled
- user imposed sequential consistency

Sequential consistency is defined as a set of operations that seem to be performed in some serial order consistent with the program order. In contrast, user-imposed consistency can be yielded due to the program order or calls to MPI_FILE_SYNC.

int MPI_File_set_atomicity (MPI_File fh, int flag)

INOUT	fh	file handle
IN	flag	true to set atomic mode, false to set non-atomic mode

Description:
Consistency semantics are guaranteed by a collective call of all processes in one group. In detail, any read or write operation to a file can be regarded as an atomic operation.

Features:

- collective call (values for *fh* and *flag* must be the same for all processes in one group)

int MPI_File_get_atomicity (MPI_File fh, int flag)

IN	fh	file handle
OUT	flag	true if atomic mode, false if non-atomic mode

Description:
Returns either true or false according to the atomicity.

int MPI_File_Sync (MPI_File fh)
INOUT fh file handle

Description:
All previous writes to *fh* by the calling process are updated. Furthermore, all updates of other processes are visible as well. Thus, a subsequent read which is executed by the calling process returns the actual data in the file rather than any "dirty read".

Features:

- collective routine

Example: This example demonstrates consistency of a file which is written by process 0 and read by process 1. In order to guarantee that process 1 does not read any wrong data the so-called sync-barrier-sync construct is used for the following reason [9]:

- MPI_Barrier ensures that process 0 writes to the file before process 1 reads the file
- MPI_File_sync guarantees that the data written by all processes is transferred to the storage device.
- The second MPI_File_sync ensures that all data which has been transferred to the storage device is visible to all processes.

```
if (myid==0)
/* process 0 write to the file */
{
  MPI_File_open (MPI_COMM_WORLD, "ufs:file2", MPI_MODE_CREATE |
    MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
  MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native",
    MPI_INFO_NULL);

  for (i=0; i<1000; i++)
```

```
    buf1[i]=i;

  MPI_File_write (fh, buf1, 1000, MPI_INT, &status);

  MPI_File_sync(fh);
  MPI_Barrier (MPI_COMM_WORLD);
  MPI_File_sync(fh);

  MPI_File_close(&fh);
}
else
/* other processes read the updated file */
{
  MPI_File_open (MPI_COMM_WORLD, "ufs:file2", MPI_MODE_CREATE |
    MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
  MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native",
    MPI_INFO_NULL);

  MPI_File_sync(fh);
  MPI_Barrier (MPI_COMM_WORLD);
  MPI_File_sync(fh);

  MPI_File_read (fh, buf1, 1000, MPI_INT, &status);
  MPI_File_close(&fh);
}
```

Although the second sync seems to be redundant, omitting it would yield an erroneous program.

Chapter 4

ViPIOS Interface

4.1 The ViPIOS Approach

Conventionally two different directions in developing programming support are distinguished: Runtime libraries for high-performance languages (e.g. Passion [14]) and parallel file systems, (e.g. IBM Vesta [7]).

We see a solution to the parallel I/O problem in a combination of both approaches, which results in a dedicated, smart, concurrently executing runtime system, gathering all available information of the application process both during the compilation process and the runtime execution. Initially it can provide the optimal fitting data access profile for the application and may then react to the execution behavior dynamically, allowing to reach optimal performance by aiming for maximum I/O bandwidth.

This approach led to the design and development of the Vienna Input Output System, ViPIOS ([4, 5]).

ViPIOS is an I/O runtime system, which provides efficient access to persistent files, by optimizing the data layout on the disks and allowing parallel read/write operations. ViPIOS is targeted as a supporting I/O module for high performance languages (e.g. HPF).

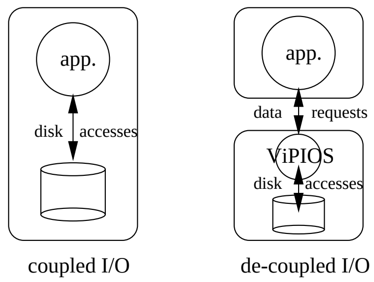

Figure 4.1: Disk access de-coupling

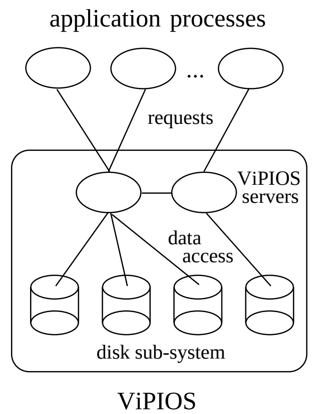

Figure 4.2: ViPIOS system architecture

4.2 System Architecture

The basic idea to solve the I/O bottleneck in ViPIOS is *de-coupling*. The disk access operations are de-coupled from the application and performed by an independent I/O subsystem, ViPIOS. This leads to the situation that an application just sends general I/O requests to ViPIOS, which performs the actual disk accesses in turn. This idea is caught by Figure 4.1.

Thus ViPIOS's system architecture is built upon a set of cooperating server processes, which accomplish the requests of the application client processes. Each application process AP is linked by the ViPIOS interface VI to the ViPIOS servers VS (see Figure 4.2).

The server processes run independently on all or a number of dedicated processing nodes on the underlying MPP. It is also possible that an application client and a server share the same processor.

Generally each application process is assigned exactly one ViPIOS server (which is called the *buddy server* to the application), but one ViPIOS server can serve a number of application processes, i.e. there exists a one-to-many relationship between the application and the servers (see Figure 4.3). The other ViPIOS servers are called *foe server* to the application.

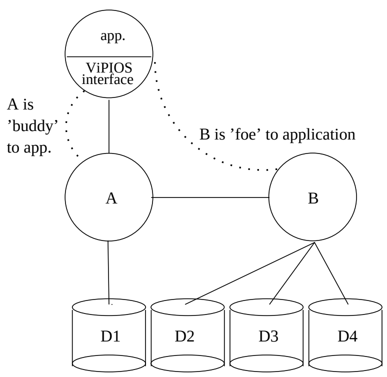

Figure 4.3: "Buddy" and "Foe" Servers

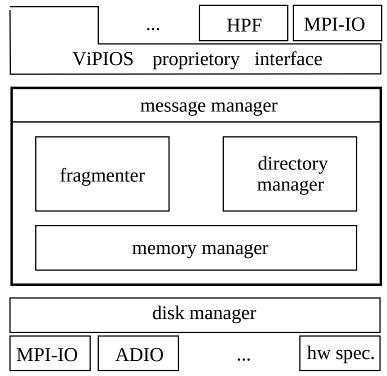

Figure 4.4: ViPIOS server architecture

4.2.1 ViPIOS server

A ViPIOS server process consists of several functional units as depicted by Figure 4.4.

Basically we differentiate between 3 layers:

- The *Interface layer* provides the connection to the "outside world"(i.e. applications, programmers, compilers, etc.). Different interfaces are supported by *interface modules* to allow flexibility and extendibility. Until now we implemented an HPF interface module (aiming for the VFC, the HPF derivative of Vienna FORTRAN) a (basic) MPI-IO interface module, and the specific ViPIOS interface which is also the interface for the specialized modules.
- The *Kernel layer* is responsible for all server specific tasks.
- The *Disk Manager layer* provides the access to the available and supported disk sub-systems. This layer too is modularized to allow extensibility and to simplify the porting of the system. At the moment ADIO [15], MPI-IO, and Unix style file systems are supported.

The ViPIOS kernel layer is built up of four cooperating functional units:

- The *Message manager* is responsible for the external (to the applications) and internal (to other ViPIOS servers) communication.

- The *Fragmenter* can be seen as "ViPIOS's brain". It represents a smart data administration tool, which models different distribution strategies and makes decisions on the effective data layout, administration, and ViPIOS actions.

- The *Directory Manager* stores the meta information of the data. We designed 3 different modes of operation, centralized (one dedicated ViPIOS directory server), replicated (all servers store the whole directory information), and localized (each server knows the directory information of the data it is storing only) management. Until now only localized management is implemented.

- The *Memory Manager* is responsible for prefetching, caching and buffer management.

Requests are issued by an application via a call to one of the functions of the ViPIOS interface, which in turn translates this call into a request message which is sent to the buddy server.

The *local directory* of the buddy server holds all the information necessary to map a client's request to the physical files on the disks. The *fragmenter* uses this information to decompose (*fragment*) a request into sub-requests which can be resolved locally and sub-requests which have to be communicated to other ViPIOS-servers (foe servers). The *I/O subsystem* actually performs the necessary disk accesses and the transmission of data to/from the AP.

4.2.2 System Modes

ViPIOS can be used in 3 different system modes, as

- runtime library,
- dependent system, or
- independent system.

These modes are depicted by Figure 4.5.

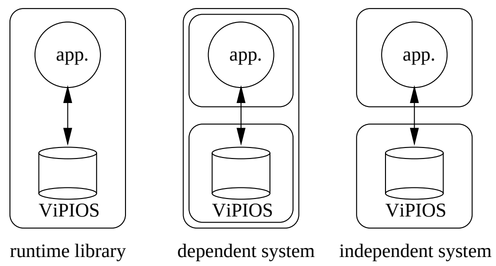

Figure 4.5: ViPIOS system modes

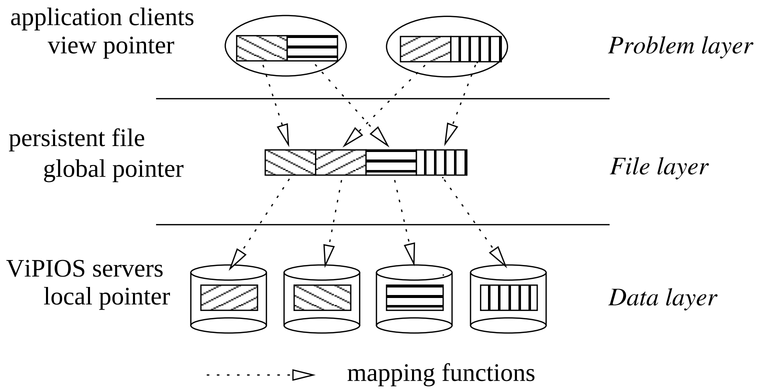

Figure 4.6: ViPIOS data abstraction

Runtime Library. Application programs can be linked with a ViPIOS runtime module, which performs all disk I/O requests of the program. In this case ViPIOS is not running on independent servers, but as part of the application. The ViPIOS interface is therefore not only calling the requested data action, but also performing it itself. This mode provides only restricted functionality due to the missing independent I/O system. Parallelism can only be expressed by the application (i.e. the programmer).

Dependent System. In this case ViPIOS is running as an independent module in parallel to the application, but is started together with the application. This is inflicted by the MPI[1] specific characteristic that cooperating processes have to be started together in the same communication world. Processes of different worlds can not communicate until now. This mode allows smart parallel data administration but objects a preceeding preparation phase.

Independent System. This is the mode of choice to achieve highest possible I/O bandwidth by exploiting all available data administration possibilities. In this case ViPIOS is running similar to a parallel file system or a database server waiting for applications to connect via the ViPIOS interface. This connection is realized by a proprietary communication layer bypassing MPI. We implemented two different approaches, one by using PVM, the other by patching MPI. A third promising approach is just evaluated by employing PVMPI, a possibly uprising standard under development for coupling MPI worlds by PVM layers.

4.3 Data Abstraction in ViPIOS

ViPIOS provides a data independent view of the stored data to the application processes.

Three independent layers in the ViPIOS architecture can be distinguished, which are represented by file pointer types in ViPIOS.

[1]The MPI standard is the underlying message passing tool of ViPIOS to ensure portability

- Problem layer. Defines the problem specific data distribution among the cooperating parallel processes (View file pointer).
- File layer. Provides a composed view of the persistently stored data in the system (Global file pointer).
- Data layer. Defines the physical data distribution among the available disks (Local file pointer).

Thus data independence in ViPIOS separates these layers conceptually from each other, providing mapping functions between these layers. This allows *logical data independence* between the problem and the file layer, and *physical data independence* between the file and data layer.

This concept is depicted in Figure 4.6 showing a cyclic data distribution.

4.4 ViPIOS Interface

In this section we describe the interface of the ViPIOS which serves as the basis for the MPI-IO implementation we will describe in the next chapters. The first part deals with establishing a connection to the ViPIOS and how to disconnect at a later stage. In the second part we summarize the commands to perform file manipulation with the ViPIOS.

4.4.1 Connection and Disconnecting

Before an application program can use the functionality of the ViPIOS a connection must be established first.

bool ViPIOS_Connect(int ViPIOS_System_ID)

IN	ViPIOS_System_ID	system identifier

Description:
Initializes the ViPIOS and establishes a connection between an application program and the ViPIOS.

Example:
```
ViPIOS_Connect(0);
```

bool ViPIOS_Disconnect(void)

Description:
Disconnects the application program from the ViPIOS.

bool ViPIOS_Shutdown(void)

Description:
Shuts the ViPIOS down. All processes are closed and the ViPIOS FAT is written back. This function can only be used by an administrative interface.

4.4.2 File Manipulation

bool ViPIOS_Open(const char filename[], int flags, int *fid)

IN	filename	name of the file
IN	flags	file access mode (R, W, RW)
INOUT	fid	file identifier

Description:
Opens an existing or creates a new file.

Example:
```
ViPIOS_Open ("matrix", 'r', &fid1);
```

bool ViPIOS_Close(int fid)

IN	fid	file identifier

Description:
Closes an open file.

bool ViPIOS_Remove(const char filename[])

IN	filename	name of the file

Description:
Removes an existing ViPIOS file.

int ViPIOS_File_set_size (int fid, int size)

INOUT	fid	file identifier
IN	size	size (in bytes) to truncate or expand file

Description:
Resizes the file defined by *fid*.

int ViPIOS_File_get_size (int fid, int *size)

IN	fid	file identifier
OUT	size	size of the file in bytes

Description:
Returns the current size in bytes of the file defined by *fid*.

4.4.3 Data Access

Blocking Routines

bool ViPIOS_Read (int fid, void *buffer, int count, int offset)

IN	fid	file identifier assigned in ViPIOS_Open
OUT	buffer	initial address of buffer
IN	count	number of bytes to read from file
IN	offset	byte offset

Description:
Reads data from an open file denoted by the file identifier into *buffer*. The last parameter *offset* states whether the operations is a so-called routine with *explicit offset* or not. Further information is given in the next routine.

Example:
```
ViPIOS_Read (fid1, buf, 15,-1);
```

int ViPIOS_Read_struct (int fid, void *buffer, int len, Access_Desc *desc, int offset, int at)

IN	fid	file identifier assigned in ViPIOS_Open
OUT	buffer	initial address of buffer
IN	len	number of bytes to read from file
IN	desc	initial address of access descriptor
IN	offset	displacement of the file
IN	at	offset relative to the displacement

Description:
Reads data from an open file in a strided way according to the file access pattern, i.e. the file view, specified by *desc*. *offset* is similar to the parameter *disp* in the routine *MPI_File_set_view* and specifies the start position of the strided access. The last parameter *at* allows distinguishing between data access with explicit offset and data access with an individual file pointer. The value -1 means that the file is read from the current position. Any value greater than 0 sets the file pointer to the specified position. However, since data access with explicit offsets should not interfere with data access with individual file pointers (see corresponding section in the chapter about MPI-IO), the file pointer is not updated after the read operation. Thus, the file pointer is only updated if the value of the parameter *at* is set to -1. A detailed description of how the strided access is accomplished and how the parameter *desc* is used is given in the next section.

Example:
```
ViPIOS_Read_struct (fid1,buf,40,view_root,20,-1);
```

40 byte values starting from the position 20 are read according to the file access pattern defined by *view_root*. Since the last parameter (*at*) is set to -1, the data access with an individual file pointer is simulated. Thus, the file pointer is updated.

```
ViPIOS_Read_struct (fid1,buf,40,view_root,20,80);
```

Here, data access with explicit offset is simulated. The file is read from position 20 relative to the beginning of the file. Furthermore, the file is read from position 80 relative to the file access pattern. In contrast to the previous example the file pointer is not updated. The exact meaning of the two different offsets becomes clear when we take a look at the implementation of the MPI-IO routines.

bool ViPIOS_Write (int fid, const void *buffer, int count, int offset)

IN	fid	file identifier assigned in ViPIOS_Open
IN	buffer	initial address of buffer
IN	count	number of bytes read from file
IN	offset	byte offset

Description:
Writes data contained in *buffer* to an open file denoted by the file identifier. The last parameter *offset* states whether the operations is a so-called routine with *explicit offset* or not.

Example:
```
ViPIOS_Write (fid1, buf, 15,-1);
```

int ViPIOS_Write_struct (int fid, const void *buffer, int len, Access_Desc *desc, int offset, int at)

IN	fid	file identifier assigned in ViPIOS_Open
IN	buffer	initial address of buffer
IN	len	number of bytes to read from file
IN	desc	initial address of ViPIOS access descriptor
IN	offset	displacement of the file
IN	at	offset relative to the displacement

Description:
Writes data in a strided way according to *desc* to an open file starting from position *disp*. The parameters *offset* and *at* have the same meaning as in *ViPIOS_Read_struct* we have analyzed above.

Example:
```
ViPIOS_Write_struct(fid1,buf,40,view_root,120,-1);
```

Non-Blocking Routines

int ViPIOS_Iread (int fid, void *buffer, int count, int offset, int *req_id)

IN	fid	file identifier assigned in ViPIOS_Open
OUT	buffer	initial address of buffer
IN	count	number of bytes to read from file
IN	offset	byte offset
IN	req_id	identifier of the request

Description:
Reads data from an open file denoted by the file identifier into *buffer* in a non-blocking way.

Example:
```
ViPIOS_Iread (fid1, buf, 15,-1,&req_id);
```

int ViPIOS_Iread_struct (int fid, void *buffer, int len, Access_Desc *desc, int offset, int at, int *req_id)

IN	fid	file identifier assigned in ViPIOS_Open
OUT	buffer	initial address of buffer
IN	len	number of bytes to read from file
IN	desc	initial address of access descriptor
IN	offset	displacement of the file
IN	at	offset relative to the displacement
IN	req_id	identifier of the request

Description:
Reads data from an open file denoted by the file identifier into *buffer* in a non-blocking way.

int ViPIOS_Iwrite (int fid, void *buffer, int count, int offset, int *req_id)

IN	fid	file identifier assigned in ViPIOS_Open
IN	buffer	initial address of buffer
IN	count	number of bytes to read from file
IN	offset	byte offset
IN	req_id	identifier of the request

Description:
Writes data contained in *buffer* to an open file denoted by the file identifier in a non-blocking way.

Example:
`ViPIOS_Write (fid1, buf, 15,-1,&req_id);`

int ViPIOS_Iwrite_struct (int fid, const void *buffer, int len, Access_Desc *desc, int offset, int at, int *req_id)

IN	fid	file identifier assigned in ViPIOS_Open
IN	buffer	initial address of buffer
IN	len	number of bytes to read from file
IN	desc	initial address of ViPIOS access descriptor
IN	offset	displacement of the file
IN	at	offset relative to the displacement
IN	req_id	request identifier

Description:
Writes data in a strided way according to *desc* to an open file starting from position *disp*.

int ViPIOS_File_Test (int req_id, int *flag)

IN	req_id	identifier of the request
OUT	flag	flag

Description:
This routine checks whether an outstanding non-blocking routine has finished. The result is given in *flag*.

Example:
`ViPIOS_File_Wait (req_id, &flag);`

int ViPIOS_File_Wait (int req_id);

int ViPIOS_File_Test (int req_id)

IN	req_id	identifier of the request

Description:
This routine checks waits until an outstanding non-blocking routine has finished.

Example:
`ViPIOS_File_Wait (req_id);`

Further Access Routines

bool ViPIOS_Seek (int fid, int offset, int offset_base)

IN	fid	file identifier assigned in ViPIOS_Open
IN	offset	absolute file offset
IN	offset_base	update mode

Description:
Updates the file pointer of a file according to *offset_base*, whereas following features are possible:

- SEEK_SET: pointer is set to *offset*
- SEEK_CUR: pointer is set to the current pointer position plus *offset*
- SEEK_END: pointer is set to the end of file

Example:
`ViPIOS_Seek (fid1, 50, SEEK_SET);`

The file pointer is set to position 50 of the file denoted by the file identifier.

bool ViPIOS_Seek_struct (int fid, int offset, int offset_base, Access_Desc *desc)

IN	fid	file identifier assigned in ViPIOS_Open
IN	offset	absolute file offset
IN	offset_base	update mode
IN	desc	initial address of ViPIOS access descriptor

Description:
Updates the file pointer of a file according to *offset_base* within a predefined file access pattern rather than merely in a contiguous way.

Example:
`ViPIOS_Seek (fid1, 50, SEEK_SET, view_root);`

The file pointer is set to position 50 of the file according the file access pattern, i.e. file view.

int ViPIOS_File_get_position (int fid, int *pos)

IN	fid	file identifier
OUT	pos	position of file pointer

Description:
Returns the current position of the individual file pointer in bytes relative to the beginning of the file.

4.5 How to use the ViPIOS

4.5.1 Quick Start

On describing the interface of the ViPIOS we will now explain all steps which are necessary to use the ViPIOS runtime library from an application program written in MPI. We assume that the ViPIOS server has already been compiled and the library *libvipios.a* resides in the same directory. This library contains the interface we described in the previous section.

First, the application program must be compiled and linked with the ViPIOS library. The syntax is the same as for a usual C or FORTRAN compiler. For example,

```
gcc -o vip_client application1.c libvipios.a
```

Thus, the application program *application1.c* is treated as a client process called *vip_serv*.

Next, the application schema must be written. This is a text file which describes how many server and client processes you want to use and on which host they should run. A possible application schema *app-schema* for one server and one client process is:

```
vipios2 0 /home/usr1/vip_serv
vipios1 1 /home/usr1/vip_client
```

In that example the server process *vip_serv* is started on the host called *vipios2* whereas the client process *vip_client* is started on the host *vipios1*.

4.5.2 An Example Program

Let us first analyze a simple program which opens a file called *infile*, reads the first 1024 bytes of the file and stores them in a file called *outfile*. Further assume that the program is run by one server and one client process.

The client program *application1.c* looks like follows:

```
#include <stdio.h>
```

```
#include "mpi.h"
#include "vip_func.h"

void main ( int argc, char **argv )
{
  int i,fid1, fid2;
  char outfile [15], buf[1024];

  MPI_Init (&argc, &argv);

  ViPIOS_Connect (0);
  ViPIOS_Open ("infile", 'r', &fid1);
  ViPIOS_Read (fid1, (void *) buf, 1024);
  ViPIOS_Close(fid1);

  ViPIOS_Open (outfile, 'w', &fid2);
  ViPIOS_Write (fid2, (void *) buf, 1024);
  ViPIOS_Close(fid2);

  ViPIOS_Disconnect();
  ViPIOS_Shutdown();
}
```

The next step is to specify the number of servers and clients which should be involved in the computation. As we stated before we want to run 1 server and 1 client process. Thus, we define a text file called *app11-schema* which contains following information:

```
vipios1 0 /home/usr1/vip_serv
vipios2 1 /home/usr1/vip_client
```

We assume that the server and the client program reside in the specified directories. Furthermore, we see that the server process *vip_serv* is started on *vipclus1* and the client process on *vipios2*. We are now ready to start the application program as we described previously.

4.5.3 Strided Data Access

In this section we will describe how a file can be accessed in strided way rather than in contiguous chunks as it is true, for example, for *ViPIOS_Read*. The approach analyzed here refers to the routines *ViPIOS_Read_struct, ViPIOS_Write_struct ViPIOS_Seek_struct*. Thus, it is possible to define a certain view to a file similar to the function *MPI_File_set_view*. As a consequence, the application program can access the data as if it were contiguous although it is physically scattered across the whole file. Let us now analyze the underlying data structure and how it can be applied for accessing a file in a strided way:

```
typedef struct
{
  int                   no_blocks;
  struct basic_block    *basics;
  int                   skip;
}
Access_Desc;

struct basic_block
{
  int                    offset;
  int                    repeat;
  int                    count;
  int                    stride;
  Access_Desc            *subtype;
  int                    sub_count;
  int                    sub_actual;
};
```

Basically, the data structure which we refer to as the "ViPIOS access descriptor" consists of two structs whereas the first struct *Access_Desc* defines the number of blocks *no_blocks* and a displacement *skip* which can be used similar to the parameter *disp* in *MPI_File_set_view*. The second struct *basic_block* specifies each block.

In order to understand the functionality of this data structure let us assume that we wish to access a file according to the view in the Figure 4.7.

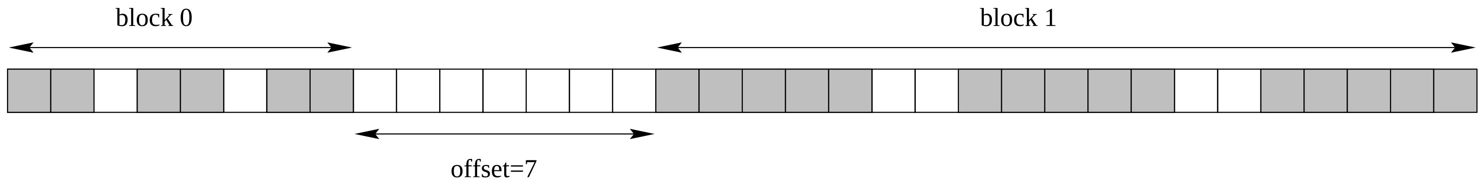

Figure 4.8: Two different file views with vectors

in a more heterogeneous way. Assume that the first part of the file should be accessed in the way described in the previous section whereas the second part should be accessed differently. The whole view is depicted in picture Figure 4.8.

The access pattern of the second part of the file corresponds to the vector

```
MPI_Type_vector(3,5,7,MPI_BYTE,&vector2);
```

Since the file is accessed in two different patterns we set *no_blocks* to 2. The second basic block is defined as: *repeat=3*, *count=5*, *stride=2*. Since the gap between the first access pattern (basic block 0) and the second access pattern (basic block 1) is 7 elements, we set *offset* to 7.

Taking a look at the definition of the ViPIOS access descriptor we notice the pointer *subtype* which is a pointer to the struct *Access_Desc*. Thus, it can be used for more complex views. Up to now we used basic MPI datatypes for accessing our files. In particular, the parameter *oldtype* of *MPI_Type_vector* was MPI_BYTE. Now assume that we define a nested datatype such that *oldtype* of the second derived datatype is in turn a derived datatype rather than a basic one. For example,

```
MPI_Type_vector (2,5,10, MPI_BYTE, &level1);
MPI_Type_vector (3,2,3,level1, &level2);
```

The derived datatype *level1* describes the pattern depicted in Figure 4.9.

Since *oldtype* of the derived datatype *level2* is the derived datatype *level1*, the access pattern described by *level2* is depicted in Figure 4.10.

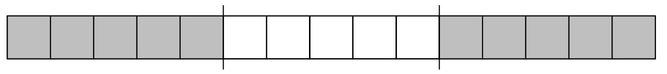

Figure 4.9: Level 1

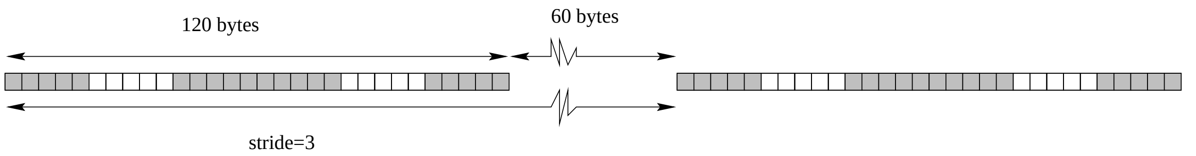

Figure 4.10: Level 2

level2 consists of 3 blocks where the size of each block is 2. In order to describe this nested access pattern with the ViPIOS access descriptor *subtype* is needed which points to another level of access pattern. More information is given when we describe the implementation of *MPI_File_set_view*.

Chapter 5

ViMPIOS: Implementation of MPI-IO

5.1 Introduction to ViMPIOS

ViMPIOS (Vienna Message Passing/Parallel Input Output System) is a portable, client-server based MPI-IO implementation on the ViPIOS. At the moment it comprises all ViPIOS routines currently available. Thus, the whole functionality of ViPIOS plus the functionality of MPI-IO can be exploited. However, the advantage of ViMPIOS over the MPI-IO proposed as the MPI-2 standard is the possibility the assign each server process a certain number of client processes. Thus, the I/O can actually be done in parallel. What is more, each server process can access a file scattered over several disks rather than residing on a single one. The application programmer need not care for the physical location of the file and can therefor treat a scattered file as one logical contiguous file.

At the moment four different MPI-IO implementations are available, namely:

- PMPIO - Portable MPI I/O library developed by NASA Ames Research Center
- ROMIO - A high-performance, portable MPI-IO implementation developed by Argonne National Laboratory
- MPI-IO/PIOFS - Developed by IBM Watson Research Center

- HPSS Implementation - Developed by Lawrence Livermore National Laboratory as part of its Parallel I/O Project

Similar to ROMIO all routines defined in the MPI-2 I/O chapter are supported except shared file pointer functions, split collective data access functions, support for file interoperability, error handling, and I/O error classes. Since shared file pointer functions are not supported, the MPI_MODE_-SEQUENTIAL mode to *MPI_File_open* is also not available.

In addition to the MPI-IO part the derived datatypes *MPI_Type_subarray* and *MPI_Type_darray* have been implemented. They are useful for accessing arrays stored in files [**?**].

What is more, changes to the parameters *MPI_Status* and *MPI_Request* have been made. ViMPIOS uses the self defined parameter *MPIO_Status* and *MPI_File_Request*. Unlike ROMIO, the parameter textsfstatus can be used for retrieving particular file access information. Thus, *MPI_Status* has been modified. The same is true for *MPI_Request*. Finally, the routines *MPI_Wait* and *MPI_Test* are modified to *MPI_File_wait* and *MPI_File_test*.

At the moment, file hints are not supported by ViMPIOS yet. Using file hints would yield following advantages: The application programmer could inform the server about the I/O workload and the possible I/O patterns. Thus, complicated I/O patterns where data is read according to a particular view and written according to a different can be analyzed and simplified by the server. What is more, the server could select the I/O nodes which suit best for the I/O workload. In particular, if one I/O node is idle whereas the other deals with great amount of data transfer, these unbalances could be solved.

5.2 Initializing and Finalizing MPI-IO

On giving the information on MPI-IO and the ViPIOS-interface we will now have a look at how the ViMPIOS is implemented. In this section we want to describe the internal handling of establishing a connection to the ViPIOS server in order to use the functionalities of MPI-IO. According to [9] no explicit routine is required for initializing and finalizing an MPI-IO session. Thus, a call to *MPI_Init()* must establish a connection to the ViPIOS server as well. Moreover, a call to *MPI_Finalize()* has to disconnect from the ViPIOS server. In order to guarantee this, every C application program using MPI-IO needs to include the header file *vip_mpio_init.h* which looks like follows:

```
#define MPI_Init(argc,argv) MPIO_Init_C(argc,argv);
#define MPI_Finalize() MPIO_Finalize();
```

The routines are defined in the file *vip_mpio.c*:

```
int MPIO_Init_C(int *argc,char ***argv)
{
    return (   ((MPI_Init(argc,argv)==MPI_SUCCESS) &&
                 ViPIOS_Connect(0)) MPI_SUCCESS : -1 );
}

int MPIO_Finalize(void)
{
    return ( (ViPIOS_Disconnect()&&(MPI_Finalize()==MPI_SUCCESS))
              ? MPI_SUCCESS : -1 );
}
```

Note that this is only true for C applications. A discussion on using Fortran application programs is given in a later section.

5.3 File Management

In this chapter we will analyze how the file information is handled. In particular, each MPI-IO file contains certain information about the filename, the file handle, the access mode etc. All that information is stored by means of following struct:

```
typedef struct {
    MPI_Comm comm;
    char *filename;

    int ViPIOS_fh;
    Access_Desc *view_root,
```

```
        *descriptor;
    bool view_is_set;

    MPI_Offset disp;
    MPI_Datatype etype;
    MPI_Datatype filetype;

    int contig_filetype;
    int access_mode;
    int atomicity;
    bool already_accessed;
} File_definition;
```

The first entry to that struct is the communicator *comm*. When we recall the syntax of the routine *MPI_File_open* we find out that each file is opened by a group of processes which are referred to by a communicator. Thus, it is possible to determine whether a file is opened by only one process - this is true if MPI_COMM_SELF is used as an argument in *MPI_File_open* - or whether the file is opened by a set of processes. This information becomes vital for collective routines which we will discuss later on.

The next two parameters are the name of the file *filename* and the file handle *ViPIOS_fh*. Latter is assigned by the routine *ViPIOS_Open*. *view_root* is a pointer to the structure *Access_Desc* which is defined in the kernel of ViPIOS. The purpose of that structure is to access a file in strides. Furthermore, the variable *view_is_set* specifies whether a view to the file currently opened is set or not. We will explain the variables *view_root* and *descriptor* in more detail when we discuss the implementation of MPI_FILE_VIEW.

Disp, *etype* and *filetype* hold the information about the file view. The variable *contig_filetype* defines whether the filetype of the view is contiguous. In other words, there are no so-called holes in the file which cannot be accessed by an application. Thus, no additional algorithm is required for computing non-accessible parts of the file.

Access_mode contains the access rights to a file, e.g. read only, write only, etc. The last variable *already_accessed* states whether any file operation was carried out on the file. This becomes important when the displacement of the file view has to be computed.

In order to guarantee that an application can open several files at any time, the information of each file must be administrated carefully, i.e. when an application accesses a file with a certain file handle, the interface must be able to retrieve the information on that particular file. ViMPIOS uses a library which is also applied within the ViPIOS kernel. The idea is as follows: When a new file is opened, a table is created which is a dynamic array of integer values. Each index of that table is a pointer to the structure *File_Definition* which we analyzed in the previous section. Let us briefly explain how this table is maintained when a file is opened, accessed and closed.

In order to access our structure *File_defintion* a new variable must be defined. This is done in the function *MPI_File_open*:

```
File_definition help_fh;
```

In the next step the table which administrates all files must be created:

```
static bool first;

if (first) {
    first=FALSE;
    std_tab_init(10,5,sizeof(File_definition),&File_table);
}
```

First is a static variable which determines whether the function *MPI_File_open* is called for the first time. According to that result the table is initialized by *std_tab_init()*. The first parameter states that storage space for 10 entries to the table is allocated. The second parameter defines how much storage space is allocated at a later stage. In other words, storage space for 10 elements of the size sizeof(File_definition) is allocated at first. When the table is full, storage space for another 5 elements is allocated etc. The name of the table is denoted by the last parameter of that routine, namely *File_table*.

Let us assume that a file is correctly opened by:

```
ViPIOS_Open(filename,amode,&help_fh.ViPIOS_fh);
```

This means that the ViPIOS server determines a file handle which is stored in *help_fh.ViPIOS_fh*.

Once storage space of the table is allocated by the routine *std_tab_init()* and the structure *File_definition* is filled with values, i.e. the file name, the access mode etc. are assigned, the file handle and the corresponding file information can be added to the file administration table. This is done by the following routine:

```
std_tab_append (&File_table, &help_fh, fh);
n_open_files++;
```

This routine adds a new element to the table *File_table*. *help_fh* is a pointer to the element which has to be inserted. In our case it is the pointer to the struct *File_defintion* which holds the information on the current file (filename, file handle). The parameter *fh* can be regarded as the index of the table which can be used as a key to retrieve information on the file. Note that *fh* is not the actual file handle which is returned by the *ViPIOS_Open* call but the index to the table. The actual file handle of the file is retrieved by *help_fh->ViPIOS_fh* as we will see later on. The variable *no_open_files* holds the information about the number of files currently opened.

Let us take a look at a small example in order to explain the functionality of the routines discussed so far. Assume that an application program opens three files. As we have already stated at an earlier stage, the file handle for each file is returned by the *ViPIOS_Open* function call. For our example assume the file handles *help_fh->ViPIOS_fh* 45,46,47. On initializing the file table, those three values can be added. Thus, index 0 points to the file referred to by file handle *help_fh->ViPIOS_fh*=45. Index 1 of the file table denotes the file with the file handle 46 and so on. It is important to mention that the file handles which are used by the application are these indices rather than the actual file handles defined by *ViPIOS_Open*.

On explaining the creation of the file table we will now discuss how information about a certain file can be retrieved. Assume that our application program wants to read data from the file with the actual file handle *ViPIOS_fh*=46. According to our previous explanation the application program uses the file handle *fh* 1, i.e. the second index of our table which points to the file with the file handle 46, rather than the actual file handle *help_fh.ViPIOS_fh*. Thus, before the data of the file with the file handle 46 can be read following steps are necessary. The code can be found in *MPI_File_read(fh,...)*:

```
File_defintion *help_fh;
std_tab_get(File_table, fh, (void **) &help_fh);
```

The function *std_tab_get* returns the element defined by the index *fh*. In our case *fh*=1 which points to the file handle 46. A correct call to *ViPIOS_Read* looks like follows:

```
ViPIOS_Read(help_fh->ViPIOS_fh,...);
```

Thus, the ViPIOS routine is called with the actual file handle *help_fh->-ViPIOS_fh* rather than the entry to the file table.

To sum up, all so-called MPI-IO routines use the index of the file table as their file handle. The actual file handle is derived from the file table and is only used when the ViPIOS routines are called. Moreover, every time an MPI-IO function other than *MPI_File_open* is called, the function *std_get_table()* must be evoked in order to retrieved the information about a particular file.

When the application program closes a file, the entry to the file table must be deleted as well which is done in the routine *MPI_File_close*. However, before the file table entry can be deleted, the function *std_tab_get()* must be called in order to determine which entry has to be deleted. The index is stored in *fh*. On closing the file with a call to the routine *ViPIOS_close* the routine *std_tab_del()* is called:

```
std_tab_del(&File_tab, *fh);
n_open_files--;
```

This routine simply deletes the element with the index *fh* from the file table and decrements the counter *no_open_files* which holds the number of file currently opened.

Finally, the whole file table is removed and the allocated storage space is freed when the last open file was closed:

```
if (n_open_files==0)
   std_tab_clean(&File_table);
```

Besides creating, filling and closing the file management table, a further routine is implemented. Before any data access on a file can be performed, a special routine checks whether a corresponding file table entry exists. Thus, following routine is called

```
if (std_tab_used(File_table)==0)
{
    printf("\nMPI_File_read: File does not exist!");
    return -1;
}
```

5.4 File Manipulation

In this section we want to discuss how the routines for file manipulation are implemented in the ViMPIOS. Let us start off with *MPI_File_open*. Besides managing the file table we described in the previous chapter, this routine checks whether the restrictions for the collective mode are obeyed. In particular, files are opened in a collective way when the communicator is not MPI_COMM_SELF. Thus, if this is true, a message is broadcast to all members of the communicator group in order to check whether all processes use the same filename and access mode.

In addition to closing a file, *MPI_File_close* removes a file if the access mode MPI_MODE_DELETE_ON_CLOSE is set:

```
if (amode & MPI_MODE_DELETE_ON_CLOSE)
    ViPIOS_Remove(filename);
```

The routines *MPI_File_delete*, *MPI_File_set_size* *MPI_File_get_size* and *MPI_File_get_amode* need no further explanation since they simply call the corresponding ViPIOS interface routines, namely *ViPIOS_Remove*, *ViPIOS_File_set_size* and *ViPIOS_File_get_size*. The routine *MPI_File_preallocate* has basically the same functionality as the routine *MPI_File_set_size*. The only difference is that the file size is not truncated if the value for preallocating memory space is smaller than the actual file size.

The code fragment of *MPI_File_get_group* for returning the communicator group of the specified file is:

```
int MPI_File_get_group(MPI_File fh, MPI_Group *group)
{
    ...
    return MPI_Comm_group(help_fh->comm, group);
}
```

Finally, the routine *MPI_File_get_amode* returns the access mode of a particular file which is currently opened:

```
int MPI_File_get_amode (MPI_File fh, int *amode)
{
    ...
    *amode=help_fh->access_mode;
    ...
}
```

5.5 File Views

In this chapter we want to describe how an MPI-IO view is realized in ViMPIOS. Before any view can be set a derived datatype has to be specified, for example

```
MPI_Datatype vector1;

MPI_Type_vector (5,2,10, MPI_INT, &vector1);
MPI_Type_commit(&vector1);
```

Next the view can be set

```
MPI_File_set_view(fh, 0, MPI_INT, vector1, "native",
MPI_INFO_NULL);
```

In our example we know the our filetype *vector1* is a derived datatype MPI_TYPE_VECTOR. Now we want to extract the variables *count*, *blocklen*, *stride* and *oldtype* in order to map the datatype to the ViPIOS access descriptor. In particular we have to search for the structure which stores the information of MPI derived datatypes. One way to do so is to take a look at the MPI implementation of the derived datatype MPI_TYPE_VECTOR. However, we will omit printing the whole source code but only analyse its content. The interested reader is refored to the lines of code of the MPI implementation.

On checking for bad arguments the derived datatype is checked for being contiguous. This is true if *blocklen* and *stride* have the same value or if *count* is 1. Then, the derived datatype *MPI_Type_vector* can be reduced to MPI_TYPE_CONTIGUOUS. Moreover, each *MPI_Type_vector* is reduced to *MPI_Type_vector*. Thus, the stride is evaluated in bytes rather than in multiples of *oldtype*.

Since *extent* is a variable of *old_dtype_ptr* which in turn is a pointer to the struct *MPIR_DATATYPE* we have to search for the definition of that structure. In the source code of MPI it can be found in the header file *datatype.h* which describes all MPI datatypes, i.e. basic datatypes as well as derived datatypes:

```
struct MPIR_DATATYPE {
    MPIR_NODETYPE dte_type; /* type of datatype element
                               this is */
    MPIR_COOKIE             /* Cookie to help detect valid
                               item */
    int          committed; /* whether committed or not */
    int          is_contig; /* whether entirely contiguous */
    int              basic; /* Is this a basic type */
    int          permanent; /* Is this a permanent type */
    MPI_Aint        ub, lb; /* upper/lower bound of type */
    MPI_Aint real_ub, real_lb; /* values WITHOUT TYPE_UB/
                               TYPE_LB */
    int             has_ub; /* Indicates that the datatype has
                               a TYPE_UB */
    int             has_lb; /* Indicates that the datatype has
```

```
                                a TYPE_LB */
    MPI_Aint        extent; /* extent of this datatype */
    int               size; /* size of type */
    int           elements; /* number of basic elements */
    int          ref_count; /* nodes depending on this node */
    int              align; /* alignment needed for start of
                               datatype */
    int              count; /* replication count */
    MPI_Aint        stride; /* stride, for VECTOR and HVECTOR
                               types */
    MPI_Aint      *indices; /* array of indices, for (H)INDEXED,
                               STRUCT */
    int           blocklen; /* blocklen, for VECTOR and HVECTOR
                               types */
    int         *blocklens; /* array of blocklens for (H)INDEXED,
                               STRUCT */
    struct MPIR_DATATYPE *old_type,
                **old_types,
                *flattened;
    MPI_Datatype self;      /* Index for this structure */
#ifdef FOO
    MPI_Datatype old_type;  /* type this type is built of,
                               if 1 */
    MPI_Datatype *old_types;/* array of types, for STRUCT */
    MPI_Datatype flattened; /* Flattened version, if available */
#endif
};

extern void *MPIR_ToPointer ANSI_ARGS(( int ));

#define MPIR_GET_DTYPE_PTR(idx) \
    (struct MPIR_DATATYPE *)MPIR_ToPointer( idx )
```

Let us pick out the variables which are important for the datatype *MPI_Type_-hvector*. *dte_type* holds the information about the kind of datatype. In our example the variable contains the data MPIR_HVECTOR. *committed* states whether the derived datatype was committed in the application program by

MPI_Type_commit. Since the stride in our datatype is greater than the number of elements (*stride*=10 > *blocklen*=2), our datatype is not contiguous (*is_contiguous=false*). Furthermore, our datatype is no basic MPI datatype (*basic=false*). *count* holds the number of blocks and *blocklen* the number of elements of each block. We will explain the further variables when we actually need them for our implementation.

Since we have found the structure which holds the information about all MPI datatypes we can start our discussion about the implementation of *MPI_File_set_view*. First we define a pointer to that structure for retrieving the information of the access pattern stored in *filetype* of *MPI_File_set_view*:

```
struct MPIR_DATATYPE *view;
view=MPIR_GET_PTR(filetype);
```

Thus, the data access pattern stored in *filetype* which can be any combination of MPI basic or derived datatypes is determined by analyzing the struct MPIR_DATATYPE.

Apart from analyzing the *filetype* we also have to tell the ViPIOS about the file view. This means the information retrieved from MPIR_DATATYPE has to be mapped to the ViPIOS access descriptor which is handled by the routine *get_view_pattern*. Before this function can be called, storage space for the ViPIOS access descriptor has to be allocated and *next_free* which is defined as *void *next_free* must be set accordingly:

```
descriptor=malloc(1024);
next_free=descriptor;
```

5.5.1 The Mapping Function *get_view_pattern*

*int get_view_pattern(struct MPIR_DATATYPE *view, Access_Desc *descriptor, void **free_space)*

get_view_pattern is a recursive routine which extracts the information of *filetype*, i.e. the access pattern of the view, and maps it to the ViPIOS access descriptor. The recursion is called as long as the *filetype* is no basic MPI datatype. Furthermore, the return value is 1 if the access pattern is contiguous and 0 otherwise. First, the function checks whether the *filetype* is one of the following derived datatypes:

- MPI_TYPE_CONTIGUOUS
- MPI_TYPE_HVECTOR
- MPI_TYPE_HINDEXED
- MPI_TYPE_STRUCT

In the chapter about derived datatypes we stated 6 different derived datatypes rather than 4. The function *get_view_pattern* checks only for four different ones since the datatype *MPI_Type_vector* is automatically reduced to *MPI_Type_hvector*, the same is true for *MPI_Type_indexed* and *MPI_Type_hindexed*.

MPI_TYPE_CONTIGUOUS

Let us start with the simplest derived datatype and explain how it is mapped to the ViPIOS access descriptor.

MPI_Type_contiguous(count,oldtype,newtype)

count holds the number of elements of the contiguous datatype. This value is retrieved by *view->count*. Thus, it can be mapped as follows:

```
descriptor->no_blocks=1;
descriptor->skip=0;
descriptor->basics[0].offset=0;
descriptor->basics[0].repeat=1;
if (!next->basic)
    descriptor->basics[0].count=view->count;
else
    descriptor->basics[0].count=view->count*next->extent;
descriptor->basics[0].stride=0;
```

Since we use just one homogenous datatype *no_blocks* and *repeat* are set to 1. What is more, *skip* and *offset* are set to 0. Some more explanation must be given for the parameter *count* which is set according to *next->basic* where *next* is an auxiliary variable for retrieving information of *oldtype*:

```
struct MPIR_DATATYPE *next;
next=view;
next=next->oldtype;
```

Assume a file which consists of 100 double values and we wish to define the following view:

```
MPI_Type_contiguous(10,MPI_DOULBE,&contig1);
MPI_Type_commit(&contig1);
MPI_File_set_view(....,contig1,...);
```

On applying the function *get_view_pattern* to that example the kind of datatype which is recognized is MPI_TYPE_CONTIGUOUS. The kind of datatype of *oldtype* which is MPI_DOUBLE can only be retrieved by our auxiliary variable *next* which points to the next level in the structure of the filetype. Thus, *next->basic* is true since *oldtype* is a basic datatype, namely MPI_DOUBLE. In other words, *next->basic* only returns *true* if the filetype does not consist of nested derived datatypes.

Moreover, since every access operation in ViPIOS is done in bytes, *count* must be multiplied by the size of the datatype of *oldtype* which is retrieved by *next->extent*. In our example *view->count* is 10, thus, the variable *count* of the ViPIOS access descriptor is set to 80 - provided the case that a the datatype double comprises eight bytes.

In short, if the view is a nested derived datatype, *count* gets the same values as retrieved from *view->count*, i.e. from the MPI struct MPIR_DATATYPE otherwise *count* is multiplied by the extent of *oldtype*.

After this little excursion about the variable *count* of the ViPIOS access descriptor we can now return to the remaining variables.

```
if (!next->basic)
{
```

```
    descriptor->basics[0].subtype=*free_space;
    get_view_pattern(view->oldtype,descriptor->basics[0].subtype,
                     free_space);
}
else
{
    descriptor->basics[0].subtype=0;
    return 1;
}
```

If *filetype* is a nested derived datatype, *subtype* points to the next free space allocated for the ViPIOS access descriptor. Furthermore, the recursive function is evoked again to map the next level of *filetype*. *free_space* is evaluated as follows:

```
free_space=(struct basic_block ) ((void) descriptor+sizeof
            (Access_Desc));
descriptor->basics=*free_space;
free_space+=(descriptor->no_blocks*sizeof(struct basic_block));
```

After the first line of code *free_space* points to the beginning of the struct *basic_block* i.e. the first struct of the ViPIOS access descriptor. Since the space for the struct *Access_Desc* is skipped *descriptor->basics* is set to that position as well. In the third line *free_space* is adjusted, i.e. the space for one basic block is skipped since *no_blocks* is 1.

MPI_TYPE_HVECTOR

On explaining the mapping mechanism of MPI_TYPE_CONTIGUOUS we can now resume with the next derived datatype, namely *MPI_Type_vector*:

*int MPI_Type_hvector(count, blocklength, stride, oldtype, *newtype)*

The values can be retrieved by *view->count*, *view->blocklen* and *view->stride*. Similar to the previous datatype only one basic block is required and *count* is set to *view->blocklen*. In order to avoid any confusion we refer to the variables of the MPI struct MPIR_DATATYPE as *view->count*, *view->blocklen* etc. Thus, when we state, for example, *count* we refer to the ViPIOS access descriptor. Moreover, note that *repeat* corresponds to *view->count* and *count* to *view->blocklen*. This is the reason why *count* is mapped to *view->blocklen* rather than to *view->count*:

```
descriptor->no_blocks=1;
descriptor->skip=0;
descriptor->basics[0].offset=0;
descriptor->basics[0].repeat=view->count;
...
if (!next->basic)
    descriptor->basics[0].count=view->blocklen;
    get_view_pattern(view->oldtype,descriptor->basics[0].subtype,
                     free_space);
else
    descriptor->basics[0].count=view->blocklen*next->extent;
```

In addition to assigning the values for *count* the recursive function is called again since the *filetype* is a nested datatype. However, in the program code the recursion is called after the mapping of the remaining values. Here we changed the order of the program code to some extent so that we can more easily explain the mapping technique.

As we have already stated in a previous chapter *stride* and *view->stride* are interpreted in a different way. *stride* denotes the gap between two data blocks of the vector whereas *view->stride* denotes the number of bytes from the beginning of one block to the beginning of the next one. *stride* is computed as follows:

```
stride=view->stride - view->blocklen * next->extent;
```

where *next->extent* holds the extent of *oldtype*. For example,

```
MPI_Type_hvector(2,5,40,MPI_INT,&vetor1);
```

The corresponding view is depicted in Figure 5.1:

The stride between the first and the second data block is *view->stirde*=40 which is mapped as:

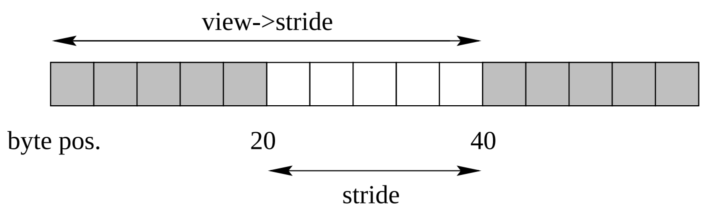

Figure 5.1: MPI_Type_hvector

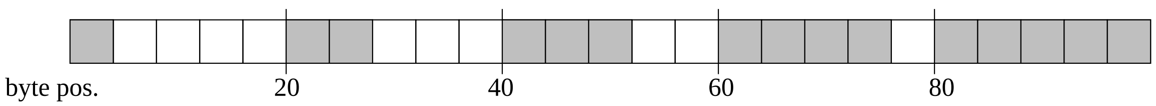

Figure 5.2: MPI_Type_indexed

```
stride=40-5*4=20;
```

since the extent of *oldtype*, i.e. MPI_INT, is 4. *Subtype* and the remaining auxiliary variables are adjusted as before.

MPI_TYPE_HINDEXED

Unlike the previous datatypes MPI_TYPE_HINDEXED is mapped by using several basic blocks since each data block can have a different size:

*int MPI_Type_hindexed (count, *array_of blocklengths, *array_of_displacements, oldtype, *newtype);*

The values can again be retrieved by *view->count*, *view->blocklens[i]* and *view->indices[i]* where i ranges from 0 to *view->count-1*, i.e. *view->blocklen[2]* states the length of the 3rd data block with a stride of *view->indices[2]*.

Since we have 5 data blocks of different lengths we use 5 basic blocks (*no_blocks= view->count* where *count*, i.e. the size, of each block is:

```
if (!next->basic)
    descriptor->basics[i].count=view->blocklens[i];
```

```
else
    descriptor->basics[0].count=view->blocklens[i]*next->extent;
```

Next *repeat* is set to 1 and *stride* to 0. Finally, the offset for each basic block must be computed because the array *view->indices* denotes the displacement of each data block to the beginning of the datatype rather than the stride between two adjacent blocks. For example, the byte displacement of the second block is *view->indices[1]*=20. The gap between the second and the third block is computed by:

*gap[2]=view->indices[2] - view->blocklens[1]*extent_of_oldtype - view->indices[1]=40 - 2*4 - 20 = 12*

The offset for the first basic blocks is:

```
descriptor->basics[0].offset=view->indices[0];
```

All remaining offsets are computed similar to the simplified formula given above:

```
if (!next->basic)
    descriptor->basics[i].offset=view->indices[i] - descriptor->
    basics[i-1].count*next->extent - view->indices[i-1];
else
    descriptor->basics[i].offset=view->indices[i] - descriptor->
    basics[i-1].count - view->indices[i-1];
```

This needs some more explanation. Let us start with the fist case were *next->basic* is false, i.e. the view is a nested derived datatype. In this case *count* of the previous basic block (*descriptor->basics[i-1].count*) is multiplied by the extent of *oldtype* since *count* was not adjusted before. On the other hand, if the view is no nested derived datatype (*next->basic=true*) *count* was already adjusted, i.e. the size of that datatype was already multiplied by the extent of *oldtype*(*count=view->blocklens[i]*next->extent*). Thus, the size of the previous basic blocks needs no longer be multiplied by the extent of *oldtype* in order to compute the correct offset.

Recalling our example of the datatype MPI_TYPE_HINDEXED we want to compute the offset of the third basic block which is 12, i.e. the stride between the second and the third basic block. Further assume that our view is no nested datatype. Thus, the length of the previous basic blocks is already adjusted (it is 8 rather than 2, i.e. 2*extent of integer=8). The offset is computed according to the second case:

offset=40 - 8 - 20 = 12

The difference to our previous computation is that the length of the previous block is already adjusted (8) and needs no more modification.

MPI_TYPE_STRUCT

The most complex derived datatype has the following syntax:

*int MPI_Type_struct(count, *array_of_blocklentghs, *array_of_displacements, oldtype, *newtype)*

Like we did for MPI_TYPE_HINDEXED we use several basic blocks, namely:

```
descriptor->no_blocks=view->count;
descriptor->skip=0;
```

count is also defined in the same way. However, the recursive function call differs to some extent. Since each block can consist of different datatypes, the first parameter of the function call is not *view->oldtype* but *view->oldtypes[i]* where i refers to the corresponding block. Thus, we use another auxiliary pointer and adjust it for each basic block:

```
next=view;
next=next->oldtypes[i];
...
if (!next->basic)
{
    descriptor->basics[i].count=view->blocklens[i];
    ...
    get_view_pattern(view->oldtypes[i],descriptor->basics[0].
    subtype,free_space);
}
```

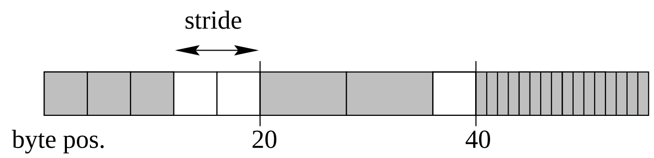

Figure 5.3: MPI_Type_struct

```
else
    descriptor->basics[0].count=view->blocklens[i]*next->extent;
```

Again, *repeat* is set to 1 and *stride* to 0. More detailed information must be given about the mapping of *offset*.

The offset of the first basic block is:

```
descriptor->basics[0].offset=view->indices[0]
```

whereas the remaining offsets are computed as follows:

descriptor->basics[i].offset=view->index[i]- view->blocklen[i-1] extent_of_oldtypes[i-1]-view->indices[i-1]*

Let us explain this by an example. Assume a file which consists of 5 integer, 2 double, and 50 character values. Further assume following derived datatype:

```
s_types[3]={MPI_INT,MPI_DOUBLE,MPI_CHAR};
s_blocklens[3]={3,2,16};
s_disps[0]=0;
s_disps[1]=20;
s_disps[2]=40;
```

The corresponding file view is depicted in Figure 5.3.

The values of the displacements are: *view->indices[0]*=0, *view->indices[1]*=20 and *view->indices[2]*=60. The corresponding offsets for each basic block of the ViPIOS access descriptor are computed as follows:

descriptor->basics[0].offset=0
*descriptor->basics[1].offset=20 - 3*4 - 0 = 8*
*descriptor->basics[2].offset=60 - 2*8 - 20 = 24*

When, for example, the offset of the third basic block is computed, extent of the previous *oldtype* must be retrieved. Thus, we need a further auxiliary variable:

```
prev=view;
prev=prev->oldtypes[i-1];
```

The actual code for computing the offset will be omitted here.

Up to now we assume that the *etype* of the view is a basic MPI datatype. However, in our introductory chapter we stated that an *etype* can be a derived datatype as well. Thus, a similar routine for extracting the information of the *etype* is required. The corresponding function is *get_oldtype*. To start off, the a pointer to the structure which stores the information on MPI datatypes is set in the routine *MPI_File_set_view*. In order to reduce computational overhead, the routine *get_oldtype* is only called if the *etype* actually is a derived datatype. Moreover, the variable *orig_oldtype_etype* gets the basic MPI datatype of the derived datatype. In other words, if the *etype* is the following vector:

```
MPI_Type_vector(10,5,20,MPI_INT,&vector}
```

orig_oldtype_etype will be MPI_INT. The remaining lines of code are an extract of the routine *MPI_File_set_view*:

```
struct MPIR_DATATYPE *oldtype_etype;

oldtype_etype=MPIR_GET_DTYPE_PTR(etype);
orig_oldtype_etype=6;

// only call the function if etype is a derived datatype
if (!oldtype_etype->basic)
    orig_oldtype_etype=get_oldtype(oldtype_etype);
else
    orig_oldtype_etype=oldtype_etype->dte_type;
```

Let us now take a look at the extraction function. We already know the features of the derived datatype such that we can quickly motivate the following lines of code. First, depending on the derived datatype, the corresponding branches are executed. Thus, the routine is called recursively until the analyzed datatype is a basic MPI datatype which is in turn returned to the calling function, namely *MPI_File_set_view*. Note that the code for the datatypes MPIR_HINDEXED and MPIR_STRUCT is more complex because these datatypes can consist of more than one oldtype.

```
int get_oldtype (struct MPIR_DATATYPE *old_datatype)
{
    struct MPIR_DATATYPE *next;
    int i;

    next=old_datatype;

    switch (old_datatype->dte_type)
    {
        case MPIR_CONTIG:
        case MPIR_HVECTOR:
        {
            next=next->old_type;
            if (!next->basic)
                get_oldtype(old_datatype->old_type);
            else
                return next->dte_type;
            break;
        }

        case MPIR_HINDEXED:
        case MPIR_STRUCT:
        {
            for (i=0; i<old_datatype->count; i++)
            {
                 next=next->old_types[i];
                 if (!next->basic)
                     get_oldtype(old_datatype->old_types[i]);
                 else
```

```
                    return next->dte_type;

            }
        }
        default:
            return 6;
    }
}
```

We still need a mechanism which checks the *etype* and *filetype* for correctness. This means, that the original oldtype of the *etype* must correspond with the original oldtype of the *filetype*. For example, if *etype* is set to MPI_INT and filetype is set to *MPI_Type_vector* its oldtype must be MPI_INT as well. Thus, one more line must be added to the routine *get_view_pattern* which is responsible for verifying these parameters.

On analyzing how a view, which is specified by means of derived datatypes, is mapped to the ViPIOS access descriptor we can now resume our discussion on the implementation of the MPI-IO function *MPI_File_set_view*.

Since this function is collective, all processes must use the same extent of *etype* and the same data representation. At the moment ViMPIOS only supports the *native* data representation. Thus, if at least two processes are started by the application program, a message is broadcast to check for the same *etypes*. If the *etypes* differ among the processes, the application program is aborted. The last step of the routine *MPI_File_set_view* is to assign following parameters:

```
ViPIOS_fh->view_root=root;
ViPIOS_fh->disp=disp;
ViPIOS_fh->etype=etype;
ViPIOS_fh->filetype=filetype;
ViPIOS_fh->info=info;
```

The last parameter *info* is not implemented yet. Finally, the file pointer is set to position 0 within the file view.

5.6 Data Access Routines

5.6.1 Read and Write

Let us start with the routine *MPI_File_read*. Unlike we pointed out in the introductory chapter on MPI-IO, the last parameter, namely *status* is of type *MPIO_Status* rather than *MPI_Status*. We will discuss its features later in this chapter.

On opening the table for the file management and checking the parameters from the application program, the next important step is to interpret the MPI datatype in a way which can be understood by the ViPIOS server. Thus, the routine *convert_datatype* is called which looks like follows:

```
void convert_datatype(MPI_Datatype datatype, int *count)
{
    switch (datatype)
    {
        case MPI_SHORT:
            (*count)*=sizeof(short int);
            break;
        case MPI_INT:
            (*count)*=sizeof(int);
            break;
        case MPI_LONG:
            (*count)*=sizeof(long int);
            break;
        case MPI_UNSIGNED_CHAR:
           (*count)*=sizeof(unsigned char);
            break;
        case MPI_UNSIGNED_SHORT:
            (*count)*=sizeof(unsigned short int);
            break;
        case MPI_UNSIGNED:
            (*count)*=sizeof(unsigned int);
            break;
        case MPI_UNSIGNED_LONG:
            (*count)*=sizeof(unsigned long int);
```

```
            break;
        case MPI_FLOAT:
            (*count)*=sizeof(float);
            break;
        case MPI_DOUBLE:
            (*count)*=sizeof(double);
            break;
       default: (*count)*=sizeof(char);
 }
}
```

This routine has two main functions. On the one hand, it interprets an MPI datatype to a C datatype, on the other hand, the parameter *count* which states the number of elements to be read is interpreted in byte elements. For example, if the application program prompts to read *count*=10 values of MPI_DOUBLE, *count* this is set to 80 provided the case that the size of one double value is 8 bytes.

After having adjusted the parameter *count*, the ViPIOS interface can be called in order to read the values. Thus, depending on the filetype of the view, either *ViPIOS_Read* or *ViPIOS_Read_struct* is called. This means that if the file view is contiguous, i.e. there are no holes in the view, the normal *ViPIOS_Read* is called with following parameters:

```
ViPIOS_Read (help_fh->ViPIOS_fh, buf, count,-1);
```

The first parameter is the entry retrieved from the file management table. The second and the third parameters refer to the buffer and the number of elements to be read. Some more information must be provided for the last parameter. -1 signals the ViPIOS server that the file pointer shall be updated after this call. Thus, if the file pointer was at position 0 before reading the file, it points to position 80 after 80 byte values were read. By means of that parameter we can explain the routine *MPI_File_read_at* which is a so-called routine with an explicit offset. Assume, the file should be read starting from position 100. The corresponding call to the ViPIOS server would be:

```
ViPIOS_Read (help_fh->ViPIOS_fh, buf, count,100);
```

Since routines with explicit offsets shall not interfere with routines without explicit offsets, the file pointer is not updated after calling that routine.

After this little excursion to another data access routine we will resume our discussion of *MPI_File_read*. We have already mentioned that depending on the file view a special ViPIOS routine is called. Thus, if the file view is not contiguous, following call is made:

```
ViPIOS_Read_struct(help_fh->ViPIOS_fh,buf,count,
    help_fh->view_root,help_fh->disp,-1);
```

The parameter *help_fh->view_root* is a pointer to the structure which handles the access pattern of the file. *help_fh->disp* holds the information about the displacement of the view. The remaining parameters have the same meaning as we discussed above.

Although we have already talked about *MPI_File_read_at* we still need to say something about the offset. Remember that every offset in MPI-IO is interpreted in units of *etype*. Thus, if our *etype* is, for instance, MPI_DOUBLE, the offset must be multiplied by the extent of MPI_DOUBLE. Restricting *etypes* to basic MPI datatypes allows us to deploy the function *convert_datatype* we discussed above. The routines *MPI_File_read_all* and *MPI_File_read_at_all* have the same implementation features as their non-collective counterparts. The only exception is a barrier at the end of the routine which synchronizes all processes in the group. This means, that the application can only resume after all processes have executed the collective read operation.

Recall that we have stated at the beginning of the section that the parameter *status* is of type *MPIO_Status* rather than *MPI_Status*. The ViMPIOS internal structure of *MPIO_Status* looks like follows:

```
typedef struct
{
    int fid;
    int count;
}
MPIO_Status;
```

On calling the ViPIOS interface for any data access routine, for example, *ViPIOS_Read*, the number of bytes which were actually read are stored by the ViPIOS server in the structure *MPIO_Status*. Thus, the file identifier must be stored in the structure which is done in all blocking data access routines:

```
status->fid=help_fh->ViPIOS_fh;
```

The number of bytes which were actually accessed by a particular data access routine can be retrieved by the routine *MPI_File_get_count*. Since we have not discussed its synapses in the introductory chapter on MPI-IO, we will take a look at the whole program code:

```
int MPI_File_get_count (MPIO_Status *status,
    MPI_Datatype datatype, int *count)
{
    int res;

    if (status->fid<0)
    {
        printf("\nFile handle is not specified since non-
                 blocking request has not finished correctly!");
        res=0;
    }
   else
        res=ViPIOS_File_get_count(status->fid,count);

     return ( (res==1) ? MPI_SUCCESS :  -1) ;
}
```

After calling the routine *ViPIOS_File_get_count* with the ViPIOS-file handle stored in *status->fid* the parameter *count* holds the number of bytes which were actually accessed.

Besides blocking access operations ViMPIOS also supports non-blocking ones. In particular, *MPI_File_iread* and *MPI_File_iread_at* as well as *MPI_File_iwrite* and *MPI_File_iwrite_at*. However, similar to ROMIO split collective routines are not supported yet. Note that the difference between the parameters of blocking and non-blocking routines is the last parameter, namely *request*, which can be regarded as the index for the particular non-blocking routine. In addition, the routine *MPI_File_test* allows testing whether the non-blocking routine has finished. Moreover, *MPI_File_wait* is used for waiting until the routine specified by the request number has finished.

Similar to the parameter *status* of the blocking data access routines, the type of the parameter *request* of the non-blocking calls is redefined as well. Thus, instead of *MPI_Request* ViMPIOS uses the type *MPIO_Request*, which is defined by:

```
typedef struct
{
    int  reqid;    /* Request-Id */
    int  fid;      /* File-Id */
}
MPIO_Request;
```

Moreover, each non-blocking data access stores the ViPIOS-file identifier into that structure which is needed for the routines *MPI_File_wait*, *MPI_File_test* and *MPI_File_get_count*:

```
request->reqid=help_fh->ViPIOS_fh;
```

On calling a non-blocking routine, the function *MPI_File_test* can be used for checking whether the data access operation is finished which is indicated by the parameter *flag*:

```
int MPI_File_test (MPIO_Request *request, int *flag,
    MPIO_Status *status)
{
 int res;

 res=ViPIOS_File_Test(request->reqid, flag);

 if (*flag==TRUE)
     status->fid=request->reqid;
 else
        status->fid=-1;
```

```
 return ( (res==1) ? MPI_SUCCESS :  -1) ;
}
```

If the *flag* is set, i.e. the non-blocking data access operation has finished, the request identifier is stored in the structure of *MPIO_Status* otherwise the file identifier is set to -1. The reason for this functionality can be explained by looking at the parameters of *MPI_File_get_count*. Recall that the file identifier is required for determining the number of bytes which are accessed. Since this file identifier is retrieved from the parameter *status* which is not used for non-blocking operations the value must be set in the routine *MPI_File_test*. Thus, we yield two advantages. On the one hand, we can retrieve the number of bytes actually accessed, the other hand, if the non-blocking function has not finished, i.e. *status->fid* is set to -1, the ViPIOS interface *ViPIOS_File_get_count* needs not be called.

The routine *MPI_File_wait* is implemented in a similar way. However, the difference to the previous one is that it waits until the non-blocking operation specified by the *request* has finished.

```
int MPI_File_wait(MPIO_Request *request, MPIO_Status *status)
{
    int res;

    res=ViPIOS_File_Wait(request->reqid);
    status->fid=request->reqid;

    return ( (res==1) ? MPI_SUCCESS :  -1) ;
}
```

Since the implementation of the routines for writing data to a file - *MPI_File_write*, *MPI_File_write_at*, *MPI_File_write_at_all* are analogous to the read operations we will omit a comprehensive explanation.

5.6.2 Scatter Functionality

Recalling the syntax of the data access routines we assumed so far that our data type is a basic MPI data type. However, even more complicated structures can be used in order to simulate a so-called scatter function. In particular this means that a file which is contiguous in memory can be read into the read buffer in a strided way. Let us take a look at an example in order to show the difference to the conventional read operation. Assume that our file resides on the disk in a contiguous way. Further assume that no file view is set. Performing following *read* operation yields a read buffer which exactly corresponds to the data stored on disk.

```
MPI_File_read(fh, buf, 100, MPI_INT, status);
```

This data access operation simply reads 100 integer values into the read buffer. In order to simulate the scatter mechanism, a derived datatype is used rather than a basic MPI datatpype.:

```
MPI_Type_vector(2,5,50,MPI_INT,&vector);
MPI_Type_commit(&vector);

MPI_File_read(fh, buf, 1, vector, status);
```

Although the parameter *count* is set 1, more than 1 integer value is read. In particluar, 10 integer values are read since the derived datatype comprises 10 elements. However, the read buffer is not filled in contiguously but according to pattern described by the derived datatype. Thus, the first five elements of the read buffer are filled with the values which are read from the file and 45 values are skipped. Finally, the last 5 elements are read into the read buffer. On the whole, the read buffer comprises 55 elements whereas merely 10 integer values stem from the file stored on disk.

By means of that mechanism an even more complex data access is possible. Assume that you set a file view according to *filetype_vector* whereas the datatype for the read buffer corresponds to the derived datatype *read_vector*. Thus, a non-contiguous file can be stored in a different buffer which is non-contiguous as well. Let us demonstrate this case by means of an example:

```
MPI_Type_vector(3,10,20,MPI_INT,&filetype_vector);
MPI_Type_commit(&filetype_vector);

MPI_Type_vector(4,3,10,MPI_INT,&read_vector);
MPI_Type_commit(&read_vector);
```

```
MPI_File_set_view(fh, 0, MPI_INT, filetype_vector, "native",
    MPI_INFO_NULL);
MPI_File_read(fh, buf, 3, read_vector, status);
```

The same mechanism can be applied for writing a file.

5.6.3 Further Data Access Routines

The routine for updating the file pointer *MPI_File_seek* is based on *etype* units as well. Thus, the offset must be converted in the same way as we have stated for the routines with explicit offsets, for example, *MPI_File_read_at*. Next, the actual seek operation is performed where we have to distinguish between two cases. As we have already stated, the file view can either be contiguous or non-contiguous. First case means that the so-called normal seek operation does not suffice. Thus, the ViPIOS server automatically realizes that internally a *ViPIOS_Seek_struct* is called. However, the only consequence for the MPI-IO implementation is to call the ViPIOS routine *ViPIOS_Seek* such that the displacement of the view is added to the offset:

```
if (help_fh->contig_filetype)
{

    switch (whence)
    {
        case SEEK_SET:
            res=ViPIOS_Seek (help_fh->ViPIOS_fh, offset+
                             help_fh->disp,whence);
            break;
        ...
     }
}
else
    res=ViPIOS_Seek (help_fh->ViPIOS_fh, offset, whence);
```

The current position in *etype* units can be retrieved by the routine *MPI_File_-get_position* which in turn calls the routine *ViPIOS_File_get_position*. Since the value which is received from the ViPIOS server is given in bytes rather than in *etype* units, the byte value has to be converted which is done by the routine *byte_to_etype*:

```
int byte_to_etype(MPI_Datatype datatype, int count)
{
    switch (datatype)
    {
        case MPI_SHORT:
            count/=sizeof(short int);
            break;
        case MPI_INT:
            count/=sizeof(int);
            break;
        case MPI_LONG:
            count/=sizeof(long int);
            break;
        case MPI_UNSIGNED_CHAR:
            count/=sizeof(unsigned char);
            break;
        case MPI_UNSIGNED_SHORT:
            count/=sizeof(unsigned short int);
            break;
        case MPI_UNSIGNED:
            count/=sizeof(unsigned int);
            break;
        case MPI_UNSIGNED_LONG:
            count/=sizeof(unsigned long int);
            break;
        case MPI_FLOAT:
            count/=sizeof(float);
            break;
        case MPI_DOUBLE:
            count/=sizeof(double);
            break;
        default: count/=sizeof(char);
    }
 return count
}
```

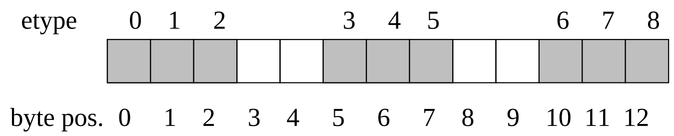

Figure 5.4: File view

This routine converts a view relative offset given in *etype* units into an absolute byte position relative to the current view. First, the offset which can be any multiple of *etype* must be converted into bytes. This is handled by the routine *convert_datatype*. If the *filetype* is contiguous, i.e. the view to the file does not contain any holes, the byte offset can be computed by:

```
*disp=offset + ViPIOS_fh->disp;
```

Otherwise the ViPIOS access descriptor which holds the information about the file view must be used to compute the byte position. Assume following file view (Figure 5.4).

In that example the offset 6 corresponds to the 11th byte position. If the view consists of nested derived datatypes, the byte position cannot be evaluated as straightforward as in that example. The routine *Fill_access_descriptor*, on the one hand, computes the extent of the view including all so called holes, i.e. the size for each block is stored in *sub_count* which is a variable of the ViPIOS access descriptor. On the other hand, it evaluates the number of elements (*sub_actual* which can actually be accessed, i.e. the size of the view without holes. Thus, the whole structure of the ViPIOS access descriptor is traversed recursively and *sub_count* as well as *sub_actual* are evaluated for each basic block.

Assume following values for the ViPIOS access descriptor:

Level 1:

```
no_blocks=1,
offset=0;
repeat=2;
count=5;
```

```
stride=5;
```

Level 2:

```
no_blocks=1;
offset=0;
repeat=3;
count=2;
stride=20;
```

Let us analyze the functionality of the recursive routine by means of the previous example. When the function is called for the first time, *cur_basic->subtype* is *true*, since our view consists of nested datatypes. *sub_count*=0 and *read_count*=0. Now the function is called recursively with the values *Fill_access_desc(cur_basic->subtype,0,0)*.

This time *cur_basic->subtype* is *false* which means that the function is not called again but *sub_count, read_count, cur_basic->sub_count* and *sub_actual* are set to 1. Furthermore, following values are computed:

```
read_sum=2 * 5 * 1 = 10;
count=0 + 2 * (5 * 1 + 5) - 5 = 15;
count=15 + 0 = 15;
act_read=10;
```

This means that the extent of the inner basic block is 15 whereas only 10 elements can actually be accessed since the stride of that access pattern is 5. However, the variables *sub_count* and *sub_actual* are 1 because no more basic blocks exist.

When the inner incarnation of the recursive function is completed, the outer incarnation is resumed after the function call. Thus, *cur_basic->sub_count*=15 and *cur_basic->sub_actual*=10, i.e. they are assigned the values which are computed by the inner incarnation and hold the information about the extent and the actual number of accessible elements of the sub basic block. Then, the extent of the whole access pattern (view) can be evaluated:

```
read_sum=3 * 2 * 10 = 60;
count=0 + 3* (2 * 15 + 20) - 20 = 130;
```

```
count = 130 + 0 = 130;
read_act=60;
```

The extent of the outer basic block is 130 whereas 60 byte elements can actually be accessed.

On preparing these values we can now derive the absolute byte position from the relative one. This is handled by the routine *Get_absolute_byte_position*. Let us again explain the functionality by means of our previous example. First, the actual size of each basic block is evaluated:

*cur_actual=cur_basic->repeat*cur_basic->count*cur_basic->sub_actual;*

In our example the outer basic block consists of 3 blocks (*repeat*=3) with the size of *count*=2. Since the view is a nested derived datatype, the size of each element is not 1 but *sub_actual*=10. Thus, *sub_actual* retrieves the number of elements of the sub basic block. The number of elements of the basic block is:

```
cur_actual= 3 * 2 * 10 = 60;
```

Next, the start offset is checked whether it lies between the range of the first basic block (*start_offset¡cur_actual*). If *start_offset* is greater than *cur_actual*, the function can only resume if a further basic block exists otherwise the position of *start_offset* would lie beyond the range of the view.

The next example will demonstrate this case. Assume that our *start_offset*, i.e. relative byte position, is 23. The corresponding absolute byte position is 53. Since 23 lies within the range of the first basic block, we resume our computation. In contrast, if our relative offset were greater than 60 the absolute byte offset would be beyond the range of the view. However, the function *Get_absolute_byte_position* assumes that the view is circular, i.e. the new start offset is computed as long as it fits into the actual length of the view:

`start_offset-=cur_actual`;

For example, if the actual length of our view is *cur_actual*=60, the total extent *sub_count*=130 and our absolute start offset is 143, then the absolute byte position of 143 is computed as follows:

```
start_offset=143-60=83;
start=0 + 3 * (2 * 15 + 20) - 20 + 0 =130 (sub_count);
...  start_offset=83-60=23;
start=130 + 3 * (2 * 15 + 20) - 20 + 0 = 260;
...  start=260 + 53 = 313;
```

Thus, the absolute byte position of *start_offset*=143 is 303 when we assume that the view of the previous example is circular.

Let us now find out how the absolute position of *start_offset*=23 is evaluated. First, we have to find out the sub block which is referred by the offset.

```
blocks=start_offset / cur_basic->sub_actual;
23 / 10 = 2;
```

Since our view consists of *repeat*count*=3*2=6 sub blocks, we know that the absolute byte position must be in the range of the third sub block (*blocks*=2 whereas the first sub block is referenced by 0). Next we have to evaluate the byte position of the beginning of the third sub block. This is done by adding the size of the first two sub blocks to the stride between sub block 2 and 3 plus a possible offset:

```
*start= blocks*cur_basic->count + (blocks/cur_basic->count)
cur_basic->stride + cur_basic->offset;
start= 2*15 + (2/2)*20 + 0 = 50;
```

Then, the offset within the third sub block is computed:

```
start_offset%=cur_basic->sub_actual
start_offset= 23 % 10 = 3;
```

Since *cur_basic->subtype* is *true*, i.e. the view consists of a nested derived datatype and consequently further sub blocks exist, the function is called recursively with the values *start_offset*=3, *start*=50. Then, the new values are evaluated accordingly:

```
cur_actual=2*5*1=10;
blocks= 3 / 1 = 3;
start= 50 + 31 + (3/5)*5 + 0 = 53;
start_offset= 3 % 1= 3;
```

As no further sub blocks exist the variable *start* holds the absolute byte position 53.

5.7 File interoperability and Consistency semantics

The routine *MPI_File_get_type_extent* returns the extent of datatypes in the file, if the datatype is no NULL type.

The routines which handle the file consistency among parallel processes, namely *MPI_File_set_atomicity* and *MPI_File_sync* are provided by the interface but are not treated explicitly be the ViPIOS server since every data access operation in ViPIOS is atomic. Thus, non-atomic mode is not supported yet.

5.8 Advanced Derived Datatypes

In our introductory chapter on MPI we have presented several MPI derived data-types. We have already mentioned that they can consist of multiple basic datatypes located either contiguously or non-contiguously in memory. Furthermore, the datatype created by a datatype constructor can be used as an input datatype for other derived datatypes. Thus, it is possible to build nested derived datatypes. The great advantage of such constructions is that noncontiguous data can be accessed with one command rather than reading the first chunk of data, skipping the data not required, reading the next chunk and so forth. In this section we will present two further derived datatypes which are part of the MPI-2 standard but not implemented in MPICH-1.1.

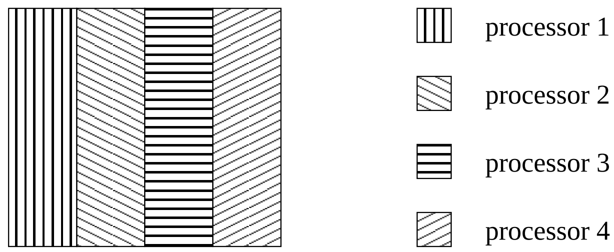

Figure 5.5: Subarray

5.8.1 Subarray Datatype Constructor

MPI_Type_create_subarray (int ndims, int *array_of_sizes, int *array_of_subsizes, int *array_of_starts, int order, MPI_Datatype oldtype, MPI_Datatype *newtype

IN	ndims	number of array dimensions
IN	array_of_sizes	number of elements of oldtype in each dimension of the full array
IN	array_of_subsizes	number of elements of oldtype in each dimension of the subarray
IN	array_of_starts	starting coordinates of the subarray in each dimension
IN	order	array storage order
IN	oldtype	array element datatype
OUT	newtype	new datatype

This datatype allows describing an n-dimensional subarray of an n-dimensional array whereas the subarray can be located at any position within the array. Thus, a global array can be distributed onto several processors such that each one gets a certain section of the array. Assume a 12x12 array should be distributed onto 4 processes. Further assume that each processor gets 3 consecutive columns of the array as depicted in Figure 5.5.

By the help of that example we will explain this datatype. *ndims* defines the number of dimensions of the global array. In our case this value is 2. Moreover, this parameter specifies the number of elements of the next three parameters *array_of_sizes[], array_of_subsizes[], array_of_starts[]*. The size of the global array and the subarray are defined by *array_of_sizes[]* and *ar-*

ray_of_subsizes. The location of the subarray within the global array is specified by *array_of_starts[]*. Hence, *array_of_starts[0]=0, array_of_starts[1]=3* describes the subarray of the second processor which starts at the position 0 in the first dimension and at position 2 in the second dimension. Since C and FORTRAN use different orders for addressing arrays, the order can be defined by the parameter *order* and can either be *MPI_ORDER_C*, i.e. row-major order, or *MPI_ORDER_FORTRAN*, i.e. column-major order. The remaining parameters refer to the datatype of the global array and the name of the created derived datatype.

In the following example we present how this datatype can be used with MPI-IO. In our example we assume that the master process, i.e. process with rank 0, firstly writes the data to the file before the new derived datatype can be applied. Thus, on setting the file view, each process can only access the part of the global array according to Figure 5.5.

```
MPI_File     fh;
MPI_Datatype subarray;

int
  rank,
  subarray[4][3],
  array_sizes[2],
  subarray_sizes[2],
  start_pos[2];

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);

/* specify the size of the global array and the subarray */
array_sizes[0]=12;
array_sizes[1]=12;
subarray_sizes[0]=4;
subarray_sizes[1]=3;

/* calculate location of the subarray according to the rank */
```

```
start_pos[0]=0;
start_pos[1]=rank*3;

MPI_Type_create_subarray(2, array_sizes, subarray_sizes,
    start_pos,   MPI_ORDER_C, MPI_INT, &filetype);
MPI_Type_commit(&filetype);

/* each process reads a particular part of the file */
MPI_File_open (MPI_COMM_WORLD, "ufs:array_file", MPI_MODE_RDWR,
    MPI_INFO_NULL, &fh );
MPI_File_set_view (fh, 0, MPI_INT, subarray, "native",
                   MPI_INFO_NULL);
MPI_File_read_all(fh, subarray, 4*3, MPI_INT, &status);
MPI_File_close(&fh);

MPI_Type_free(&subarray);
MPI_Finalize();
```

5.8.2 Distributed Array Datatype Constructor

The derived datatype we discussed previously allows accessing different subarrays of a global array where each subarray is regarded as one block. The following derived datatype supports HPF-like distribution patterns like BLOCK-BLOCK distribution or CYCLIC-CYCLIC distribution. Before we give a description of this derived datatype we will give a brief introduction to HPF-distribution patterns.

Basically two different distribution patterns are possible, namely BLOCK and CYCLIC. Let us start with a file which can be regarded as a one-dimensional array consisting of 16 elements. Further assume that we have a processor grid of four processors and we wish to distribute the file onto these four processors in the distribution pattern BLOCK(4). This means that each processor gets one contiguous block of the file consisting of 4 elements. The distribution pattern is depicted in Figure 5.6.

Another distribution pattern, for instance, is CYCLIC(1). This means that the data is distributed in a round robin fashion onto the four processors. In this case, processor 1 gets the first element of the file, processor 2 gets the

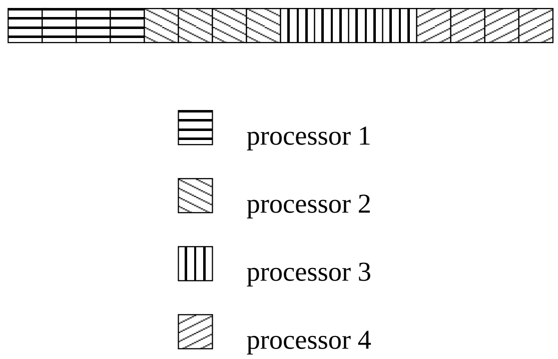

Figure 5.6: BLOCK(4)-distribution

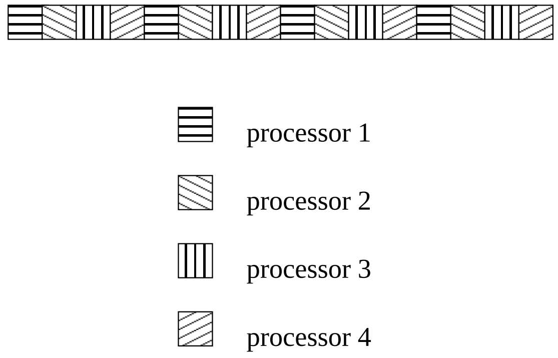

Figure 5.7: CYCLIC(1)-distribution

second element of the file and so on. When each processor has got one element, this process is repeated again. Thus, in the second run processor 1 gets the fifth element, processor 2 the sixth and so on. Figure 5.7 demonstrates the result of this distribution pattern.

Now assume that our file corresponds to a 2-dimensional array rather than a 1-dimensional one. In particular, our file should be an 8x8 array. Thus, more complex distribution patterns are possible by combining the ones we discussed so far. For example, BLOCK-CYCLIC means that the distribution pattern in the first dimension is BLOCK whereas the distribution pattern in the second dimension is CYCLIC. In order to get familiar with these pat-

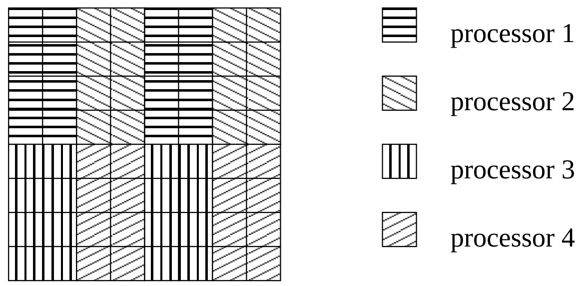

Figure 5.8: BLOCK(4),CYCLIC(2)-distribution

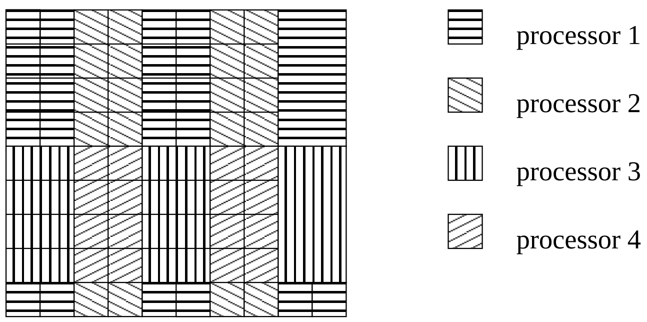

Figure 5.9: BLOCK(4),BLOCK(2)-distribution with irregular patterns

terns, we will give some examples.

In our first example we assume an 8x8 array which shall be distributed in the first dimension according to BLOCK(4) and in the second one according to CYCLIC(2). Further assume that our processor grid consists of 4 processors such that each dimension comprises two processors. The result can be seen in Figure 5.8.

In the next example we want to distribute a 9x10 array according to the pattern CYCLIC(2), CYCLIC(2). In contrast to our previous example not each processor gets the same number of elements. The result can be seen in Figure 5.9.

On giving a brief introduction to HPF-like distribution patterns, we can now resume our discussion on the derived datatype for distributed arrays.

MPI_Type_create_darray (int size, int rank, int ndims
int *array_of_gsizes, int *array_of_distribs,
int *array_of_dargs, int *array_of_psizes, int order
MPI_Datatype oldtype, MPI_Datatype *newtype)

IN	size	size of process group
IN	rank	rank in process group
IN	ndims	number of array dimensions as well as processor grid dimensions
IN	array_of_gsizes	number of elements of oldtype in each dimension of the global array
IN	array_of_distribs	distribution of array in each dimension
IN	array_of_dargs	distribution argument in each dimension
IN	array_of_psizes	size of process grid in each dimension
IN	order	array storage order
IN	oldtype	array element datatype
OUT	newtype	new datatype

By means of *MPI_Type_create_darray* a datatype can be created that corresponds to the distribution of an *ndims*-dimensional array onto an *ndims*-dimensional array of logical processes. *size* states the number of processes in the group. *ndims* specifies the number of dimensions of the array to be distributed as well as the number of dimensions of the processor grid. The size of each processor grid is given in *array_of_psizes[]*. It is important to note the following equation must be satisfied:

$$\sum_{i=0}^{ndims-1} array\ of\ psizes[i] = size$$

Let us explain this with an example. Suppose that our global array is a 2-dimensional 16x16 array and we wish to distribute it onto *size*=4 processes. Since our global array consists of *ndims*=2 dimensions, our processor grid consists of two dimensions as well. Thus, we have three different possibilities for constructing our processor grid. The first dimension could consist of *array_of_psizes[0]*=1 processor and the second of *array_of_psizes[1]*=4 (1*4=4) processors. In this case the processor grid corresponds to shape a in Figure

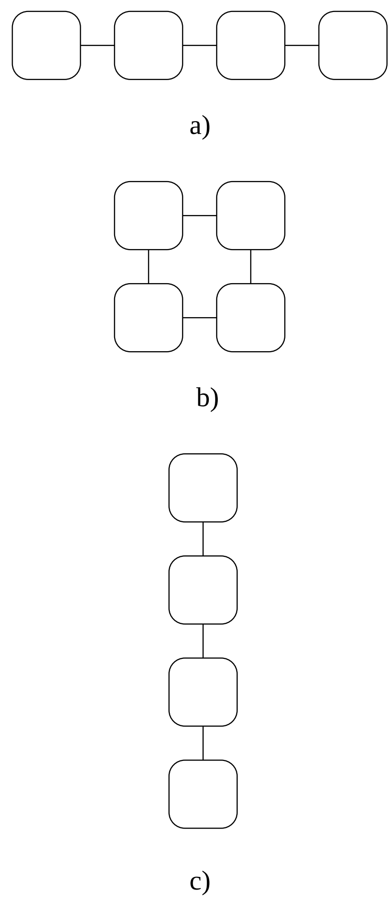

Figure 5.10: Processor grid

5.10. In shape b both dimensions comprise 2 processors. Shape c is the opposite of shape a, i.e. dimension 1 consists of 4 processors whereas dimension 2 consists of 1 processor.

This processor grid serves as the basis for the distribution pattern specified by the array *array_of_distribs[]*. In particular, each dimension of the global array can be distributed in three different ways:

- MPI_DISTRIBUTE_BLOCK- corresponds to BLOCK distribution
- MPI_DISTRIBUTE_CYCLIC- corresponds to CYCLIC distribution

- MPI_DISTRIBUTE_NONE - this dimension is not distributed

The distribution argument of each dimension is stored in *array_of_dargs[]* and can either be a number or the constant MPI_DISTIRBUTE_DFLT_DARG whereas following assumption must be satisfied:

*array_of_dargs[i] * array_of_psizes[i]>=array_of_gsizes[i].*

Let us again take a look at one example in order to see the functionality of that datatype. In particular we wish to take a look at the code fragment of an application program which distributes a 9x10 array onto four processes in the pattern CYCLCIC(2), CYCLIC(2) as we have seen in Figure 5.9. Assume that the array storage order is C.

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

ndims=2
order=MPI_ORDER_C

/* size of the array to be distributed */
array_of_gsizes[0]=9;
array_of_gsizes[1]=10;

/* distribution and distribution argument */
array_of_distribs[0]=MPI_DISTRIBUTE_CYCLIC;
array_of_distribs[1]=MPI_DISTRIBUTE_CYCLIC;
array_of_dargs[0]=2;
array_of_dargs[1]=2;

/* processor grid */
array_of_psizes[0]=2;
array_of_psizes[1]=2;

MPI_Type_create_darray(nprocs, rank, ndims, array_gsizes,
    array_of_distribs, array_of_dargs, array_of_psizes,
    order, MPI_INT; &newtype);
MPI_Type_commit(&newtype);
```

5.8.3 Implementation of the Subarray Datatype Constructor

On getting used to the functionality of the derived datatypes we can now turn to describing the implementation of *MPI_Type_create_subarray*. The basic information for the implementation of this and the following derived datatype can be found in ROMIO.

First, the input parameters to that derived datatype must be checked for being correct. In particular none of the entries for the parameters *ndims, array_of_sizes[], array_of_subsizes[]* must be less than 1. Furthermore, *array_of_subsizes[i]* must not be less than *array_of_sizes[i]*. *array_of_starts[i]* must not be less than 0 or *array_of_starts[i]* > *(array_of_sizes[i]-array_of_subsizes[i]*. Next the parameters *oldtype* and *order* are checked.

According to *order* the datatype is built by using the existing derived datatypes of MPI-1 we discussed in a previous chapter. Lets take a look at the implementation of the *order = MPI_ORDER_C*. Recall that in this case the last dimension of the array changes fastest since the ordering used by C arrays is row-major. For example, assume a 4x4 array. Thus, we address it by the indices [0][0], [0][1],[0][2],[0][3],[1][0] and so forth. The code fragment will help explaining the algorithm:

```
if (order == MPI_ORDER_C)
{
    // dimension ndims-1 changes fastest
    if (ndims == 1)
        MPI_Type_contiguous(array_of_subsizes[0], oldtype,
                            &tmp1);
    else
    {
        MPI_Type_vector(array_of_subsizes[ndims-2],
        array_of_subsizes[ndims-1],
        array_of_sizes[ndims-1], oldtype, &tmp1);

        size = array_of_sizes[ndims-1]*extent;

        for (i=ndims-3; i>=0; i--) {
```

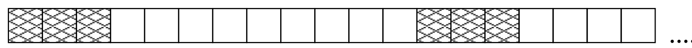

Figure 5.11: Vector

```
            size *= array_of_sizes[i+1];
        MPI_Type_hvector(array_of_subsizes[i], 1, size, tmp1,
                         &tmp2);
        tmp1 = tmp2;
    }
}
```

Next, the dimension *ndims* of the derived data is checked. If *ndims* is 1, we can reduce the whole datatype to *MPI_Type_contiguous* since the file can only be accessed in a contiguous block with the size of *array_of_subsizes[]*.

If *ndims* is greater than 1, the derived datatype can be built with the datatype *MPI_Type_vector*. Recall the example from the previous section where we distributed a 12x12 array onto 4 processes such that each process gets 3 consecutive columns of the array. Since the parameters to *MPI_Type_vector* are *count, blocklength* and *stride*, the section for the first process can be described by *MPI_Type_vector(12,3,12)*. The pattern is depicted in Figure 5.11.

In general, *count* corresponds to *array_of_subsizes[ndims-2]*, *blocklength* to *array_of_subsizes[ndims-1]* and *stride* to *array_of_sizes[ndims-1]*. If *ndims* is of an order higher than 2, a further derived datatype, namely *MPI_Type_hvector*, is used for describing the remaining dimensions.

Next, we have to add the displacement for each process. For example, process 2 should read the forth, fifth and sixth columns which is defined by *disp[1]=array_of_starts[1]=3*. Since each process is supposed to access a different block of the array, the start displacement *disps[1]* is different for each process. Finally, the derived datatype *MPI_Type_struct* is used for describing the section which should be accessed by each process. In particular, this datatype consists of three blocks whereas the second block contains the information of the access pattern we have described by *MPI_Type_contiguous*, *MPI_Type_vector* or *MPI_Type_hvector*. The first block is of the type *MPI_LB* and the third of *MPI_UB*. *MPI_LB* and *MPI_UB* are so-called "pseudodatatypes" which are used to mark the lower bound and the upper bound of a datatype. Since the extent, i.e. the span from the first to the last byte, of both datatypes is 0, they neither effect the size nor count of a datatype. The reason for using these datatypes is to define explicitly the lower and upper bound of a derived datatype. For more information we refer the reader to [8].

The implementation is shown in the following code fragment. However, a detailed description is given in the next section when we discuss the implementation of the derived datatype *MPI_Type_create_subarray*.

```
// add displacement and UB
disps[1] = array_of_starts[ndims-1];
size = 1;
for (i=ndims-2; i>=0; i--) {
    size *= array_of_sizes[i+1];
    disps[1] += size*array_of_starts[i];
}

disps[1] *= extent;
disps[2] = extent;
for (i=0; i<ndims; i++)
    disps[2] *= array_of_sizes[i];

block_lens[0] = block_lens[1] = block_lens[2] = 1;

types[0] = MPI_LB;
/* datatype we described previously */
types[1] = tmp1;
types[2] = MPI_UB;

MPI_Type_struct(3, block_lens, disps, types, newtype);
```

5.8.4 Implementation of the Distributed Array Datatype Constructor

In this section we will describe the implementation of our last derived datatype, namely *MPI_Type_create_subarray*. On checking the input parameters for correctness we can start the implementation according to the order of the array.

Similar to the previous section we will concentrate on the C order whereas we will not go into detail with describing the FORTRAN order since the assumptions are only slightly different.

Before any distribution pattern can be taken into account each process must be mapped to the processor grid. This is done according to the formula given in [9]:

```
procs=size; /* number of processes in that process group */
tmp_rank=rank;
for (i=0; <ndims; i++)
{
    procs = procs / array_of_psizes[i];
    coords[i] = tmp_rank / procs;
    tmp_rank = tmp_rank % procs;
}
```

Let us demonstrate this functionality of this formula by means of the process with rank=2. We assume that our processor grid consists of 2 processors per dimension as depicted in Figure 5.10. On applying the correct values to the formula, namely *procs=4*, *tmp_rank*=2 and *ndims*=2, we yield the result *coord[0]=1* and *coord[1]=0*. Thus, the process with rank 2 corresponds to the processor with the coordinates [0,1], i.e. the processor in the lower left corner of the processor grid.

Next, each dimension of the array is analyzed separately starting from the highest dimension. Since we will concentrate on C-order, the last dimension changes fasted. We have already stated that *MPI_Type_create_darray* allows three different ways of distributing the array, namely MPI_DISTRIBUTE_BLOCK, MPI_DISTRIBUTE_CYCLIC and MPI_DISTRIBUTE_NONE. Since latter case can be reduced to a BLOCK distribution on only one process, i.e. no distribution is actually performed, we merely have to discuss the routines which handle the first and the second case.

BLOCK-Distribution

Let us start off with the routine for the BLOCK distribution. First, the distribution argument *darg* is checked. If it is set to MPI_DISTRIBUTE_DFLT_-

DARG, the block size of the corresponding process is set to:

```
if (darg== MPI_DISTRIBUTE_DFLT_DARG)
    blksize= (global_size + nprocs-1) / nprocs;
else
    blksize=darg;
```

according to [9] whereas *global_size* corresponds to *array_of_gsizes[i]*, i.e. the size of the array in that particular dimension, otherwise *blksize* is set to the size of the distribution argument.

Next, we must check whether each process gets the same number of elements. For example, if wish to distribute a 1-dimesional array consisting of 6 elements onto 2 processors according to the pattern BLOCK(4). Thus, the process with rank 0 gets the first four blocks, whereas process 1 merely gets 2 elements. The following code fragment handles this case:

```
j=global_size-blksize*rank;
mysize=min(blksize,j);
if ( mysize < 0 )
    mysize=0;
```

Now we are ready to implement the block-distribution by means of derived datatypes we have already discussed in the chapter on MPI. In particular, we have to bear in mind the dimension of the array we are currently analyzing. Assume a 4x6 array with a distribution pattern of BLOCK(2), BLOCK(3) as depicted in Figure 5.12. This 2-dimensional array can be linearized to a 1-dimensional array as presented in the same figure. The distribution pattern of the last dimension, i.e. dimension 1, BLOCK(3) can be described by the derived datatype *MPI_Type_contiguous*. All other dimensions must be described by *MPI_Type_hvector* with the correct stride. What is more, the *oldtype* of this datatype must be the datatype of the previous dimension, i.e. a nested derived datatype is built.

In our example the distribution pattern of dimension 1 is BLOCK(3). Thus, following datatype is used:

```
MPI_Type_contiguous(3, MPI_INT, dim1);
```

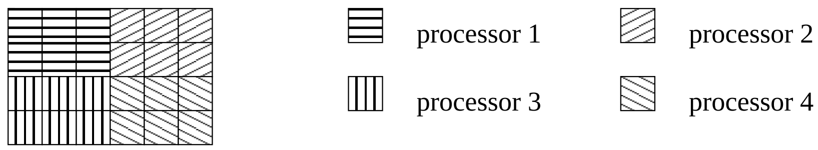

Figure 5.12: 4x6-array with BLOCK(2),BLOCK(3)-distribution

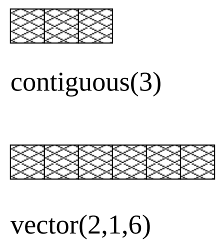

Figure 5.13: Vector distribution pattern

Since dimension 0 is distributed according to BLOCK(2), this can be described by using the derived datatype *MPI_Type_vector* whereas *oldtype* is the datatype we created before. The stride is computed as the length of the array of the next dimension, i.e. *array_of_gsize[1]*=6. Thus, the vector is:

```
MPI_Type_hvector(2,1,6,dim1,dim0);
```

The two steps which are necessary, for simulating a BLOCK(2),BLOCK(3) distribution are shown in Figure 5.13

In order to apply this algorithm for any dimension, the stride must be set accordingly, namely to:

```
for (i=nidms-1; i>dim; i--)
    stride *= array_of_gsizes[i];
```

Thus, the formula is set accordingly for a n-dimensional array.

Finally, the start offset for each process must be set. Recall that the size of the first dimension of the array to be distributed is 4 and the distribution pattern is BLOCK(2). Thus, the first process in the grid of dimension 0 should read the first and the second elements whereas the second process should start accessing the third element. In general, this is handled by the following code:

```
*st_offset = block_size * rank;
```

Let us now return to the point after calling the procedure for handling the block distribution. The loop for analyzing the distribution pattern, i.e. BLOCK, CYCLIC or NONE is repeated until the last dimension. However, each call returns a new value for the *filetype* and the corresponding offset for each process as we described above. It is important to mention that each *newtype* of the derived datatype, i.e. the distribution pattern of the particular dimension is used as the *oldtype* of the next dimension. In other words, the derived datatype of one dimension is used for building the derived datatype of the next dimension and, thus, creating a nested derived datatype which describes the distribution pattern of all dimensions. A short abstract of the code shall demonstrate this:

```
if (order==MPI_ORDER_C)
{
    for (i=ndims-1, i>=0, i--)
    {
        switch(array_of_distribs[i]
        {
            case MPI_DISTRIBUTE_BLOCK:
                Block_distribution(...,type_old, &type_new,
                                   st_offsets+i);
    break;
            case MPI_DISTRIBUTE_CYCLIC:
                Cyclic_distribution(...,type_old, &type_new,
                                   st_offsets+i);
                break;
...
         }
        /* use the new datatype as the basis for the next
           datatype * /
        type_old = type_new;
    }
```

```
    /* add displacement */
    disps[1]=st_offsets[ndims-1];
}
```

After this loop we know the access pattern (*type_new*) of the particular process and its start position *disps[1]*. The easiest way to combine all this information is by using the derived datatype *MPI_Type_struct* with the following values:

```
disps[0]=0;
disps[2]= multiple of array_of_gsizes;
block_lens[0]=block_lens[1]=block_lens[2]=1;
types[0]=MPI_LB;
types[1]=type_new;
types[2]=MPI_UB;

MPI_Type_struct(3, block_lens, disps, types, newtype);
```

In short, the distribution pattern is made up by the second block of the derived datatype. The first and the third block are made up by the datatypes MPI_LB and MPI_UB.

CYCLIC-Distribution

This section will be dedicated to the description of the second distribution pattern, namely CYCLIC. Similar to the previous section, we will merely take a look at the implementation of the C order since the implementation of the FORTRAN order only slightly differs.

First, the number of blocks per process is calculated according to a slightly modified version of the formula given in [9]:

```
nblocks = ( array_of_gsizes[dim] + blksize-1 ) / blksize;
count = nblocks / nprocs;
left_over= nblocks - count * nprocs;
if (rank < left_over)
  count = count + 1;
```

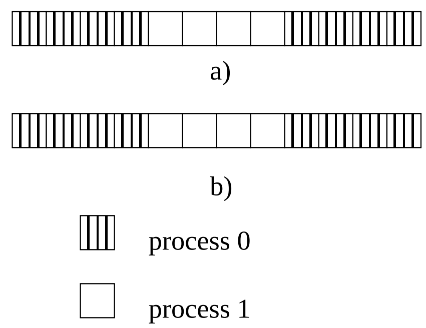

Figure 5.14: Regular and irregular distribution patterns

For example, if we wish to distribute a 1-dimesional array of 12 elements onto 2 processes with the distribution pattern CYCLIC(4) the first process gets two blocks whereas the second process only gets one block consisting of 4 elements (see Figure 5.14 a).

Next, the size of the last block is calculated. In our previous example, the size of each block was four elements. However, this is not true for an array comprising, for instance, 14 elements. Thus, each process gets two blocks but the last block of process 2 consists of only two elements rather than four (see Figure 5.14 b):

```
/* check whether any irregular block exists */
if ( (remaining_elements=array_of_gsizes[dim] %
     (nprocs*blksize)) != 0)
{
    /* compute the size of the last block */
    last_blksize=remaining_elements-blksize*rank;
        if ( (last_blksize < blksize) && (last_blksize > 0) )
        {
            count--;
            evoke_struct=1;
        }
}
```

Again, *blksize* refers to the distribution argument. In our case it is 2 -

CYCLIC(2). Note that if the size of the last block is less than the remaining ones, *count* is decremented by 1 and *evoke_struct* is set to 1. Thus, the last block is treated by means of the special derived datatype as described later on.

The regular blocks are treated in the following way. Let us explain each step by means of the introductory example where we have taken a look at the source code of an application program. Recall that the array size of the 2-dimensional array is 9x10, the distribution pattern is CYCLIC(2), CYCLIC(2) and the processor grid comprises 2 processes per dimension.

Since in the C order the last dimension changes fastest, we handle this case first. The second dimension consists of 10 elements. Thus, process 0 gets *count*=3 blocks of the size *blksize*=2 and process 1 gets 2 blocks of *blksize*=2.

```
if (dim == ndims-1)
{
    stride = nprocs*blksize*orig_extent;
    MPI_Type_hvector(count, blksize, stride, type_old, type_new);
}
```

Since the stride is given in bytes, it must be multiplied by the datatype of elements of the array. All other dimensions cannot be treated as straightforwardly. Therefore, we have to split up the access pattern into two derived datatypes, one which builds a sub block using the block size as the first parameter, and a second one which uses the number of blocks, i.e. *count*, as the first parameter. Moreover, it is a nested datatype based on the previously created sub block. In addition, the strides for these two derived datatypes must be computed. Let us list the code fragment before we explain its meaning:

```
/* compute sub_stride and stride */
sub_stride = orig_extent;
stride = nprocs*blksize*orig_extent;

for (i=ndims-1; i>dim; i--)
{
    sub_stride *= array_of_gsizes[i];
```

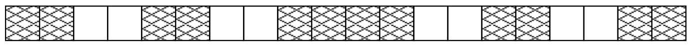

Figure 5.15: Distribation pattern of dimension 1

```
    stride *= array_of_gsizes[i];
}

/* datatypes for sub_block and block */
MPI_Type_hvector (blksize, 1, sub_stride, type_old, &sub_block);
MPI_Type_hvector(count, 1, stride, sub_block, type_new);
```

We will demonstrate our approach for the first process. Assume that we have already built the derived datatype for the second dimension. Since the first dimension consists of 9 elements, process 1 gets 3 blocks whereas the last blocks is smaller than the previous ones. Thus, *count* is reduced to 2. What is more, the block size of the regular blocks is 2 as well. Provided with that information we can now build the first derived datatype, which represents the sub block. Since our array comprises only two dimensions *sub_stride* is set to *array_of_gsizes[2]*=10. For simplicity let us restrict to the number of elements and thus not regard the actual number of byte elements, which would be necessary for the derived datatype. Thus, the pattern of the derived datatype which describes the sub block looks like follows (Figure 5.15):

Next the derived datatype which describes the whole access pattern is built. This is done by creating a derived datatype *MPI_Type_hvector* such that the number of blocks corresponds to *count*. What is more, the block size is 1 and the oldtype is the newtype from our previously created sub block. Finally, the stride must be set accordingly. First, the stride is multiplied by the number of processes of the particular processor grid dimension - in our case it is two - times the block size. Next, it is multiplied by the sizes of the array to be distributed. On building both derived datatypes the access pattern for process 1 depicted in Figure 5.16 is yielded.

Since the first dimension consists of 9 elements rather than 8, we still have to add the last block. Recall that its block size is smaller than the remaining one and thus has to be treated separately:

```
if ( dim == ndims-1)
```

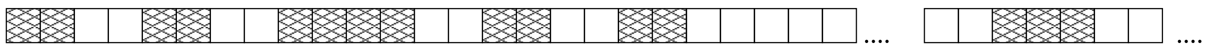

Figure 5.16: Distrubution pattern of both dimensions

```
{
    types[0] = *type_new;
    types[1] = type_old;
    disps[0] = 0;

    if (count == 0 )
        disps[1]= blksize*(nprocs-1)*sub_stride;
    else
        disps[1] = count*stride;

    blklens[0] = 1;
    blklens[1] = last_blksize;
}

else
{
    // sub_type is added!!!
    MPI_Type_hvector (last_blksize, 1, sub_stride, type_old,
                      &sub_block_struct);

    types[0] = *type_new;
    types[1] = sub_block_struct;
    disps[0] = 0;

    if (count == 0 )
        disps[1]= blksize*(nprocs-1)*sub_stride;
    else
        disps[1] = count*stride;

    blklens[0] = 1;
    blklens[1] = 1;
}
```

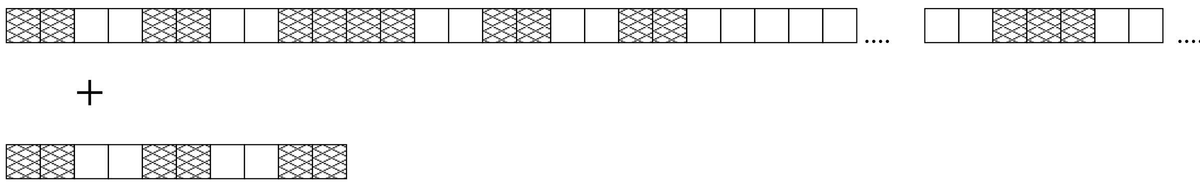

Figure 5.17: Pattern of an irregular distribution

Figure 5.18: Irregular distribution pattern of one block

```
MPI_Type_struct(2, blklens, disps, types, &type_tmp);
*type_new = type_tmp;
```

Since we are analyzing the first dimension, we have to take a look at the else branch of the outer if-statement. Again, a derived datatype for a sub block is build. Rather than taking *blksize* as the first argument, *last_blksize* is used now. Next, the previously created datatype for the regular pattern is combined with the irregular pattern by means of *MPI_Type_struct*. Note that the displacement of the irregular pattern depends on the number of blocks *count* of the regular pattern. In our case the displacement can simply be computed by multiplying the number of blocks by the stride. The new pattern is depicted in Figure 5.17.

Imagine that the first dimension of the processor grid consists of *nprocs*=4 processes rather than 2. Further assume that we have already built the derived datatype for the second dimension as shown so far. Thus, assuming a 9x10 array with a distribution pattern CYCLIC(2) the first three processes get one block consisting of two elements whereas the last process gets one block of only one element. In this case *count* is 0 and *last_blksize* is 1. Figure 5.18 demonstrates the effect of the new stride. Since *sub_stride* corresponds to the second dimension of the array, i.e. 10, *stride* is set to 2*3*10=60.

After having built the derived datatypes for the regular as well as the irregular patterns, the last step is to add the lower and upper boundary as we described for the BLOCK distribution. Although we have only discussed the implementation for a 2-dimensional array, this algorithm works for n-dimensions as well.

5.9 Fortran-Interface

In this section we want to state the most important points about converting a Fortran application program using MPI-IO to the C-interface.

The first thing to be mentioned is that every procedure call in Fortran is *call by reference*. The name of a procedure can only consist of lower case letters and must be terminated with an underscore "_". Next, strings must be treated with special care since there is no "null" which indicates the end of a character array. Thus, this indicator must be added to the character array.

In the following code fragment we will show the implementation of the MPI-IO routine *MPI_File_open*. Note that the routine is a procedure rather than a function as it is true for the C implementation. Thus, the return value from the C function is stored in the parameter **ierror*. In order to distinguish between input and output parameters, input parameters are indicated by *const*. Since the name of the file to be opened is a character array, it must be converted accordingly. First, the routine *std_str_f_to_c* is called which in turn calls the routine *strn_copy0*. On returning from both routines, the Fortran filename is converted to a character array which can correctly be interpreted by the C routine and does the actual file opening.

```
#define  STRING80_LEN 80
typedef char STRING80 [STRING80_LEN+1];


void mpi_file_open_ (const int *comm, const char *filename,
    const int *amode, const int *info, int *fh, int *ierror)
{
```

```
    STRING80 c_filename;

    /* convert a c-array into a fortran */
    std_str_f_to_c (filename, c_filename, STRING80_LEN);

    /* call the correspoding function from the c interface */
    *ierror=MPI_File_open (*comm, c_filename, *amode, *info, fh);
}

static void std_str_f_to_c (const char *fstr, char cstr [],
                            size_t max)
{
    register int i;

     /* do nothing */
     for (i=max-1; i >= 0 && fstr[i] == ' '; i--) ;

    (void) strncpy0 (cstr, fstr, i+1);
}

static char *strncpy0 (char *s1, const char *s2, size_t n)
{
    (void) strncpy (s1, s2, n);

    *(s1+n) = '\0';

    return (s1);
}
```

A further point to bear in mind when mapping from Fortran to C is the different interpretation of datatypes. For example, following MPI datatypes are supported by Fortran, which have no direct C MPI datatypes as their counterparts, namely:

- MPI_INTEGER
- MPI_REAL
- MPI_DOUBLE_PRECISION

- MPI_COMPLEX
- MPI_LOGICAL
- MPI_CHARACTER
- MPI_BYTE

Thus, the converting function, which is used in the Fortran MPI-IO interface is:

```
int MPI_Type_f2c (MPI_Datatype datatype)
{
    switch (datatype)
    {
        case MPI_CHARACTER:
            return MPI_CHAR;
            break;

        case MPI_INTEGER:
            return MPI_INT;
            break;

        case MPI_REAL:
            return MPI_FLOAT;
            break;

        case MPI_DOUBLE_PRECISION:
            return MPI_DOUBLE;
            break;

        case MPI_LOGICAL:
            return MPI_UNSIGNED;
            break;

        default: return MPI_BYTE;
    }
}
```

This datatype conversion is used, for example, in each routine for reading or writing a file. Let us therefore take a look at the mapping function for reading:

```
void mpi_file_read_ (const int *fh,void *buf,const int *count,
        const MPI_Datatype  *datatype, int *status, int *ierror )
{
    int res;
    MPIO_Status status_c;

    MPI_Datatype datatype_c;

    datatype_c = MPI_Type_f2c(*datatype);

    *ierror=MPI_File_read(*fh, buf, *count, datatype_c,
                          &status_c);

    *status=status_c.fid;
}
```

The forth parameter, namely *datatype*, is converted from the Fortran MPI representation to the C MPI representation via the routine *MPI_Type_f2c*.

We still need to focus out attention to the parameter *status* which is of type *MPIO_Status* in the C interface. Since all objects in FORTRAN are of type integer, the C interface is called with the parameter *status_c* rather than the value received from the FORTRAN application program. On returning from the C routine, the file identifier from the structure *status_c* is assigned to the parameter *status*. Thus, the FORTRAN application program merely deals with the file identifier of the particular data access routine rather than with the whole structure. The same conversion mechanism is also true for the remaining blocking data access routines.

The implementation for the non-blocking routines is analogous. However, since these routines do not contain the parameter *status*, the conversion mechanism is done for the parameter *request*:

```
void mpi_file_iread_ (const int *fh,void *buf,const int *count,
```

```
     const MPI_Datatype  *datatype, int *request, int *ierror )
{
    int res;
    MPIO_Request    request_c;
    MPI_Datatype     datatype_c;

    datatype_c = MPI_Type_f2c(*datatype);

    *ierror=MPI_File_iread(*fh, buf, *count, datatype_c,
                           &request_c);

    *request=request_c.reqid;
}
```

Thus, the application program only deals with the request identifier rather than the whole structure which must be considered in the routine for checking whether the outstanding non-blocking operation has finished:

```
 void mpi_file_test_ ( int *request, int *flag, int *status,
                       int *ierror)
{
    MPIO_Status  status_c;
    MPIO_Request request_c;

    request_c.reqid=*request;

    *ierror = MPI_File_test (&request_c, flag, &status_c);
    *status=status_c.fid;
}
```

This routine firstly assigns the request identifier received from the FORTRAN application to the structure *request_c*. Later, the value from the structure *status_c* is assigned to the parameter *status* which in turn can be used by the FORTRAN application. Similar conversions are made for the routines *mpi_file_wait_* and *mpi_file_get_count_*.

On analyzing the peculiarities of the Fortran to C interface we have to discuss the handling for connecting and disconnecting from the ViPIOS server. Before the functionalities of MPI-IO can be used, MPI-IO has to be initialized. This is done by the routine *MPIO_Init*, which is located in a Fortran module. Thus, every Fortran application program must include the module *vipmpi*. This routine, on the one hand, establishes the connection to the ViPIOS server, on the other hand, it manipulates the MPI_COMM_WORLD communicator in a way such that all client processes can use one MPI_COMM_WORLD without the interference of the server processes. In contrast to C application programs the header file similar to *vip_mpio_init.h* is not required. However, the header files *vip_mpio_f2c.h* and *vip_mpio_def.h* are still needed.

We want to conclude our discussing with a small Fortran application which uses the derived datatype MPI_TYPE_DARRAY for distributing a 4x5x6- array onto several processes according to the distribution pattern BLOCK(2), CYCLIC(3) and CYCLIC(2) whereas the processor grid consists of 2 processors per dimension. Besides the code for the application program a graphical interpretation of the distribution array is given as well. Note that all data objects in Fortran are integer values. This becomes clear when we take a look at the declaration section of the example program. For instance, the *newtype* of a derived datatype in Fortran is of the type *integer* rather than MPI_Datatype as we know it from the C application programs.

```
program main
USE vipmpi
implicit none

include 'mpif_vip.h'
include 'vip_mpio_f2c.h'
include 'vip_mpio_def_f2c.h'

character filename*81

integer newtype, i, ndims, array_of_gsizes(3)
integer order, intsize, nprocs,fh,ierr
integer array_of_distribs(3), array_of_dargs(3)
integer array_of_psizes(3)
integer readbuf(1024), writebuf(1024)
integer mynod, array_size, bufcount
```

```
      call MPI_INIT(ierr)
      call MPIO_INIT(ierr)

      call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, mynod, ierr)

      ndims = 3
      order = MPI_ORDER_FORTRAN
      filename = 'ufs:file_create'

c     specify the size of the array to be distributed
      array_of_gsizes(1)=4
      array_of_gsizes(2)=5
      array_of_gsizes(3)=6

c     distribution pattern of each dimension
      array_of_distribs(1) = MPI_DISTRIBUTE_BLOCK
      array_of_distribs(2) = MPI_DISTRIBUTE_CYCLIC
      array_of_distribs(3) = MPI_DISTRIBUTE_CYCLIC

c     distribution argument of each dimension
      array_of_dargs(1) = 2
      array_of_dargs(2) = 3
      array_of_dargs(3) = 2

      do i=1, ndims
           array_of_psizes(i) = 0
      end do

c     create processor array
      call MPI_DIMS_CREATE(nprocs, ndims, array_of_psizes, ierr)

      call MPI_TYPE_CREATE_DARRAY(nprocs, mynod, ndims,
     $     array_of_gsizes, array_of_distribs, array_of_dargs,
     $     array_of_psizes, order, MPI_INTEGER, newtype, ierr)

      call MPI_TYPE_COMMIT(newtype, ierr)
```

```
      array_size = array_of_gsizes(1) * array_of_gsizes(2) *
     $     array_of_gsizes(3)


c     write the array to the file

      call MPI_FILE_OPEN(MPI_COMM_WORLD,
     $     filename,
     $     MPI_MODE_CREATE+MPI_MODE_RDWR+MPI_MODE_UNIQUE_OPEN,
     $     MPI_INFO_NULL, fh, ierr)

      call MPI_FILE_SET_VIEW(fh, 0, MPI_INTEGER, newtype, "native",
     $     MPI_INFO_NULL, ierr)
      call MPI_FILE_WRITE_ALL(fh, writebuf, bufcount, MPI_INTEGER,
     $     status, ierr)
      call MPI_FILE_CLOSE(fh, ierr)

      call MPI_TYPE_FREE(newtype, ierr)
      call MPI_FINALIZE(ierr)
      call MPIO_FINALIZE(ierr)

      stop
      end
```

Chapter 6

Regression Suite

6.1 Program Testmpio

Up to now we were discussing the theoretical background of MPI-IO and the portable implementation on ViPIOS. Since an extensive testing phase is part of each software engineering process, we will present one test program from the University of California and Lawrence Livermore National Laboratory written in April 1998. This so-called "regression suite" verifies a lot of different MPI-IO routines by simulating different cases of application programs. For example, functions for testing collective open and close of files, independent reads and writes with file and buffer types, file control etc. Besides checking the functionalities of the MPI-IO implementation, the time for the verification procedure is taken. Since ViMPIOS does not support shared file pointers, special error handling routines and different representation modes, the last routines could not be tested.

The structure of the test program is as follows. The main function takes the input parameters, for example the user path for storing the files, and calls the routine *dotest()*, which in turn calls the different test routines which we stated before. Moreover, the time for the whole process is taken.

On giving a brief introduction do the regression suite we will now anaylise each routine separately.

6.1.1 test_manycomms

Some files are opened with a couple of different communicators. Starting with a group of processes which is split into two sub groups, reading and writing is analyzed.

In particular, following interface routines are tested:

MPI_Barrier, MPI_Comm_Free, MPI_Comm_split

Purpose:
This routine checks whether several files can be opened using nested communicators. Thus, for example, files are opened with the communicators 1,2,3,4 and 5. Then, the files are closed in a different way. For instance, the file which is opened by communicator 3 is closed first etc. In order to accomplish this task, a communicator group is split into several sub groups which operate on different nested files. What is more, consistency semantics are obeyed by using atomic access operations (*MPI_File_set_atomicity* and synchronization points (*MPI_File_sync*). However, for the time being, ViPIOS merely operates with atomic modes. What is more, the function for synchronizing concurrent file I/O is still a task to be fulfilled in the near future.

6.1.2 test_openmodes

Different open modes are checked. First, the file is opened with the access mode MPI_MODE_CREATE and MPI_MODE_WRONLY. Thus, reading this file is supposed to fail. Later, the modes MPI_MODE_READ_ONLY and MPI_MODE_DELETE_ON_CLOSE are checked.

Purpose:
This routines checks whether the open modes are consistent. In other words, a file which is opened, for example, in the *write only* mode, should not be readable and vice versa.

6.1.3 test_manyopens

First, a couple of files are opened and filled with data which are read back in a different order later on. Next, further files are opened with different access modes such that each file has a distinct entry to the file table.

Purpose:
Similar to the function *test_manycomms* the behavior of the file table is tested whether it can cope with several files which are opened at the same time and accessed in a different way.

6.1.4 test_openclose

MPI files are opened and closed in a collective way. Thus, it checks whether the MPI-IO recognizes errors with using wrong access modes for different files. What is more, a file is tried to close which was not opened before.

Purpose:
This routine merely check the basic functionality of opening and closing a file.

6.1.5 test_readwrite

Independent I/O calls are tested here. MPI_BYTE is chosen for the data access buffer as well as for the filetypes. Each node writes some bytes to a separate part of the same file which is read back later on in order to compare the results. Besides checking whether the correct number of bytes are read or written, the inbuffer and and outbuffer are compared, i.e. the input string must be identical to the output string. No file view is set in that routine.

Purpose:
This routine checks the basic functionalities of blocking data access operations.

6.1.6 test_rdwr

Collective I/O routines are tested with derived filetypes and buffer types. Thus, on the one hand a view is set, on the other hand a contiguous derived data type for the read and write buffer is used.

Purpose:
Besides checking the features of file views, the functionality of so-called data scattering is tested.

6.1.7 test_filecontrol

This routine tests different MPI-IO routines like *MPI_File_get_position*, *MPI_File_set_size* or *MPI_File_get_byte_offset*. Moreover, different file views are set. This test is concluded with splitting the communicator group into two sub groups which separately operate on the files.

Purpose:
The most important MPI-IO routines are checked.

6.1.8 test_localpointer

First, each node accesses the file in a contiguous way but with different displacements. Later, some data is skipped and a couple of byte values are written after this hole. Different read and write operations are performed in order to check the holes in the file. In the second run a file with a more complicated file view is opened in order to perform similar tests.

Purpose:
The behavior of setting file views is analyzed in detail. Besides setting correct file views, erroneous views are set to check whether the implementation recognizes these inconsistencies. A possible incorrect assignment would be to use an etype of the type MPI_INT and a filetype of the type MPI_DOUBLE.

6.1.9 test_collective

Collective I/O with explicit interleaving is tested. What is more, each process writes data of different size.

Purpose:
This routine checks the behavior of writing to a file according to different access patterns and different number of byte elements.

6.1.10 test_nb_rdwr

This routine does the same as *test_rdwr*. Except of using blocking routines it tests non-blocking operations.

6.1.11 test_nb_localpointer

This routine does the same as *test_nb_localpointer*. Except of using blocking routines it tests non-blocking operations.

Chapter 7

Test Results

7.1 The Performance of ViPIOS

In order to test the ViPIOS we carried out several experiments on a workstation cluster of 8 LINUX Pentium-133 PCs (nodes) connected by a 100 Mbit Ethernet network. Each PC had 32 MB main memory and a hard disk of 1 Gbyte. Compared to existent super computers an environment like this offers the following advantages for testing purposes:

- low hard- and software costs
- simple administration
- availability (the system can be dedicated to the test program so results are not influenced by changing workloads etc.)

Furthermore the Beowulf projects [1] [11] [12] have shown that whith some enhancements in the network topology such PC-networks can provide peak performance in excess of 1 GOPS and also high disk bandwidths.

First, we ran some scalability tests where we increased the number of server processes while keeping the number of client processes constant. Moreover we used dedicated I/O nodes which means that each process (either server or client) ran on a different node (see Figure 7.1).

Second, we experimented with non dedicated I/O nodes which means that on every node one server and one client process were running concurrently. The results forced us to differentiate between the following two cases:

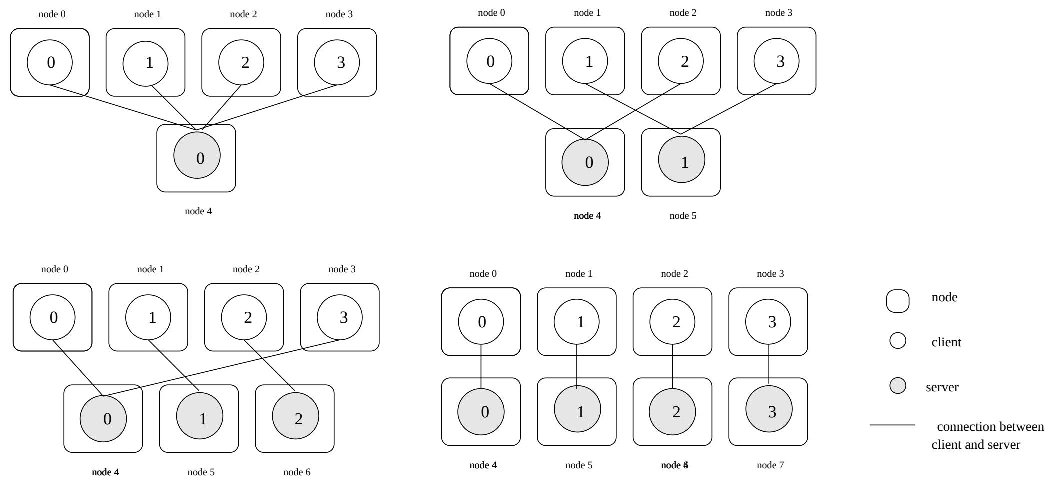

Figure 7.1: Topologies with different number of servers

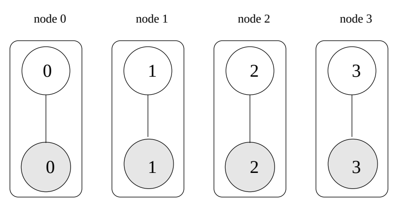

Figure 7.2: Servers and clients reside on the same node

- The client process ran on the same node as its associated server process as depicted in Figure 7.2.
- Each client process was connected to a server process on a different node as depicted in Figure 7.3.

7.1.1 Dedicated I/O Nodes

The workload for our first experiment was an 8MB file. This had to be read by four SPMD client processes (i. e. each client had to read 2 MB without overlap). In other words, client 0 read the first quarter of the file, client 1 read the second quarter and so on. As depicted in Figure 7.1 we increased

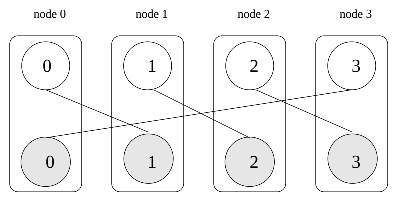

Figure 7.3: Servers and clients reside on different nodes

```
/* number of client processes is evaluated */
MPI_Comm_size(Vipios_comm_app,&nclients);

MPI_Barrier(Vipios_comm_app);
/* time is taken after all clients are synchronized */
start=MPI_Wtime();

/* each client reads a disjoint part of the file */
ViPIOS_Seek(fid1, SIZE/nclients*rank, SEEK_SET);
ViPIOS_Read(fid1,buf,SIZE/nclients);
MPI_Barrier(Vipios_comm_app);

 /* time is stopped after the last process has finished
    reading */
 end=MPI_Wtime();
```

Figure 7.4: Program code

the number of server processes from one to four.

In order to measure the overall time for reading the whole file, the concurrently running processes were synchronized before and after the read request. By imposing these two barriers we guaranteed that the time to read the whole file corresponded to the reading time of the slowest process. A short extract from the program code is shown in Figure 7.4.

The whole process was iterated 50 times. In order to suppress any caching effects data was not only read from one file but from 10 different files in a

Table 7.1: Scalability results for dedicated I/O nodes

clients	servers	max	min	mean	variance
4	1	4.32	2.09	3.02	0.0694
4	2	2.15	1.20	1.78	0.0239
4	3	2.36	1.34	1.65	0.0217
4	4	1.48	0.98	1.12	0.0115

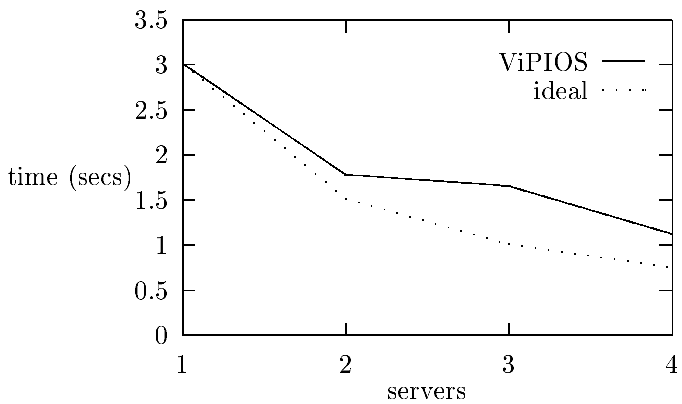

Figure 7.5: Scalability results

round robin fashion. Thus, during the first iteration file1 was read, during the second iteration file2 etc. After 10 iterations file1 was read again. It is important to state that each server read its data locally from the node it was running on.

In addition to the mean we measured the maximum, minimum and variance of the time required for reading the whole file. The results are given in Table 7.1.

The graph in Figure 7.5 depicts the mean time in seconds to read the file in relation to the number of server processes. There seems to be no significant performance gain by increasing the number of servers from two to three. But remember that clients are synchronized in order to measure the time it takes to read the 8MB file completely. In the case with three servers there is one server with two client processes attached. This server has to read and transfer the same amount of data (4MB) as any server in the two servers case. So it takes approximately the same time to accomplish the operation. The other two server-client pairs are idle almost half of the time. In fact there is a performance gain by introducing a third server but this will only show up if clients are allowed to work asynchronously.

A comparison with the ideal linear scaling (dotted line in Figure 7.5) shows that ViPIOS scales well with an increasing number of servers. Results for superior hardware configurations (MPPs or Beowulf clusters) are expected to be even better because of increased network bandwidth and less contention.

7.1.2 Non Dedicated I/O Nodes

Next we studied the results when a server process and the corresponding client process were running on the same node (see Figure 7.2). In these experiments the number of the client processes was always kept equal to the number of child processes in order to ensure that the work done on every processing node is the same.

The results were found to be strongly dependent from the topology of the ViPIOS. If the client and the server connected to it were executing on the same node (like shown in Figure 7.2) then the performance was very poor. If every client connected to a server which ran on a different node (Figure 7.3) the performance was boosted by a factor greater than 30. Further investigation showed that the reason for this strange behavior was the message passing via MPI (see 7.2.1). The performance measurements for the second case are shown in Table 7.2 for a constant workload of 8MB. That means that with 2 clients every client read 4MB, with 4 clients every client read 2MB and so on. Table 7.3 gives the values achieved when every client just read 2MB so that the actual workload varied from 4MB to 16MB.

As can be clearly seen in Figure 7.6 the ViPIOS still scales for a constant workload even if the I/O nodes are not dedicated. However the scaling factor for increasing the number of processors from 4 to 8 is very small. The graph for the increasing workload shows that the deviation from the omptimum

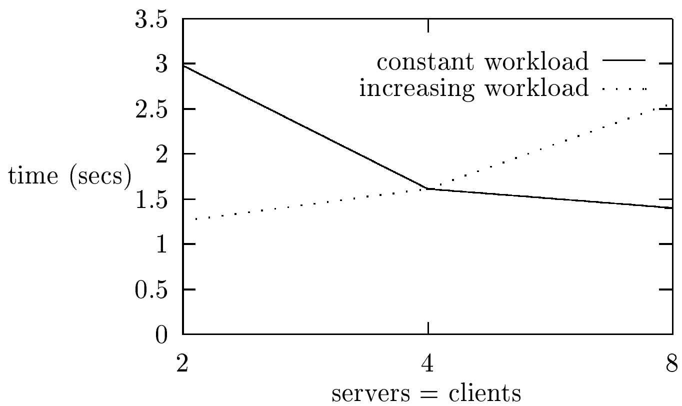

Figure 7.6: Non dedicated I/O nodes

Table 7.2: Constant workload

clients	servers	max	min	mean	variance
2	2	3.59	2.54	2.98	0.0734
4	4	1.96	1.22	1.61	0.0691
8	8	1.90	1.11	1.40	0.0413

(which would be a line parallel to the "server = client" coordinate of course) increases with the number of servers.

Both of these effects can be easily explained be the increasing network contention. By using a more elaborate network topology these problems may well be overridden. Also in a real super-computing environment the interconnection between the computing nodes will yield a better bandwidth.

Table 7.3: Increasing workload

clients	servers	max	min	mean	variance
2	2	2.08	0.97	1.26	0.0624
4	4	1.96	1.22	1.61	0.0692
8	8	2.98	2.35	2.56	0.0238

7.2 Performance Comparison With Existing I/O Systems

In order to show that the overhead imposed by message handling and file administration of the ViPIOS is low we also compared its performance to UNIX file I/O combined with MPI and to MPI-IO. Performance measurements have been taken on the PC-network described above. We used MPICH Version 1.1.0 [3] as an MPI implementation and ROMIO Version 1.0.0 [2] as an implementation of MPI-IO. All the reported values are the means of 50 measurements.

7.2.1 ViPIOS compared to UNIX file I/O and MPI

Using normal UNIX file read operations it takes 0.48 seconds to read 2MB from the local disk. To transfer 2MB of data via MPI from one node to another takes 0.52 seconds. If the sending and the receiving process reside on the same node then the data transfer via MPI takes at about 50 times as long and the variance of the duration value gets very big. This may be due to problems in synchronizing the sender and the receiver processes if they do not run independently. Here we also find the reason for the bad performance of ViPIOS when I/O nodes are non dedicated and the client process runs on the same node as the server process it is connected to.

If we compare the sum of the above times for disk read and data transfer (1.0 seconds) to the 1.12 seconds ViPIOS takes for the same operation (see 7.1) we can state that the overhead of the ViPIOS is reasonably small.

7.2.2 ViPIOS Compared to ROMIO

In order to compare ViPIOS to ROMIO we used two test setups. In the first 4 clients had to read an 8MB file using MPI-I/O operations. Since MPI-I/O does not directly support access to remote files (files which reside on an other PC than the client process) the 8MB were distributed in 2MB portions. So every client just read the 2MB from its local disk. Using ROMIO the read operation took 0.49 seconds which is in fact very close to the normal UNIX read (0.48 seconds).

This form of data distribution would be compared best to a ViPIOS with non dedicated I/O nodes where the server and the corresponding client execute on the same node. Unfortunately this is just the configuration where the performance values of ViPIOS are very bad (39.83 seconds) due to the behavior of MPI described in the previous section. To overcome this problem we optimized ViPIOS for that specific topology by using shared memory instead of MPI to transfer the data from the server to the client. After that ViPIOS just took 1.41 seconds for the complete read operation. Most of this time (0.84 seconds) was spent for request and acknowledge messages which still were transmitted via MPI. Further improvements would require to use shared memory for these messages too. Our time schedule did not allow us to implement that yet but the results above obviously imply that ViPIOS can handle I/O as quickly as MPI-I/O even for this specific topology.

In order to have a direct comparison we used a second test setup where the 8MB file was not distributed but located in a shared NFS-directory on the disk of the first PC. In this case the read operation took 3.88 seconds. The comparable ViPIOS configuration (1 dedicated server with 4 clients; see Table 7.1 excelled with 3.02 seconds.

7.3 Further Tests on ViPIOS and ViMPIOS

In the previous chapter we were discussing some tests concerning the scalability of ViPIOS. In particular we tried to show that the performance of the overall system increases when the number of sever processes is increased as. However, the upper limit for the number of server processes is determined by the number of client processes. The reason for this statement is quite obvious when we bear in mind that each client process can only be handled by one sever process rather than multiple ones.

In this chapter we will show some test results which we have already analyzed in the previous chapter. However, rather than reading file sizes of 2 MB, we will now take a look at larger file sizes. The reason for presenting these results in addition to the one we have discussed in the previous chapter is that during the first and this second test phase some changes to the ViPIOS kernel where made which could have some impacts of the overall performance. In addition to that, this workstation cluster was increased by 8 Linux PCs such that at present our system comprises 16 PC clustered together over an Ethernet network.

Later in this chapter we will compare the performance of ViMPIOS with ROMIO where we will notice that the performance of ROMIO is about 20 times better than ViMPIOS for small file sizes. The advantage can be explained by the internal caching and prefetching mechanism of ROMIO (see 2- Phase-Method). For larger file sizes this method does not improve the performance any more. Thus, the performance of ROMIO is merely three times higher for file sizes beyond 30 MB.

By means of further test series will analyze the behavior of derived datatypes. We first try to write files in a certain way, lets say BLOCK, BLOCK, and read them back at a later stage CYCLIC, CYCLIC.

We will conclude this test phase by describing the performance of what we may call *sparse derived datatypes*. A sparse derived datatype is a derived datatype which has a large *holes* between adjacent data blocks. Since ROMIO uses the 2-Phase-Method, we might expect ViMPIOS to outperform the counterpart in this special case.

7.3.1 Scalability Results With Larger Files

In this sub section we want to present to scalability results for a 60 MB file. Following assumptions are made for the analysis: The number of client processes is kept constant throughout each test phase whereas the number of server processes ranges from 1 to 4. According to our experience, which we

Table 7.4: Scalability results II for dedicated I/O nodes

clients	servers	max	min	mean	variance
4	1	88.79	81.61	86.08	10.1245
4	2	52.00	46.12	49.04	5.7647
4	3	51.91	45.89	48.52	6.3144
4	4	28.80	25.30	26.51	2.6367

derived from the tests in the previous section, we would expect the system to have a significant performance gain when the number of server processes is increased. The results are depicted in the Table 7.4.

Since the curve resembles where much the one depicted in Figure 7.5, we will not preset it here one more but refer the reader to the previous chapter.

7.3.2 ROMIO vs. ViMPIOS

Read-Write-Behavior

In the introduction of this chapter we stated that ROMIO's performance for writing a file and reading it back later on is much higher for small files. The larger the files get, the better ViMPIOS converges to the results of ROMIO although ViPIOS does not support any internal caching or prefetching algorithm to optimize the system yet. In the table we give the results for both systems. However, if the file size lies beyond 200 MB, ViPIOS still works erroneously. Thus, some more research work must be invested in order to verify this drawback. The results can be seen in Table 7.5.
In order to compare the two systems directly the second column gives the ratio of the mean times for ROMIO and ViMPIOS. The further colums state the test results of ROMIO whereas the last columns contain the equivalent information on ViMPIOS. To sum up the results very briefly we can state state ROMIO's performance outranges ViMPIOS at on order of magnitude of 21 for file sizes of about 16 MB. However, this gap shrinks to a factor of merely 2.5 for file sizes lager than 16 MB. In our table we only depict the values for three different file sizes.

Table 7.5: Performance of ROMIO vs ViMPIOS

File size	R. vs. V.	max	min	mean	max	min	mean
16	1:21	5.19	1.36	1.95	43.15	41.17	41.09
40	1:2.50	167.23	42.79	105.94	339.36	239.09	289.31
100	1:2.50	406.80	154.62	279.60	702.12	696.36	698.81

Derived Datatypes

This kind of tests is supposed to be the most promising ones for ViMPIOS due to the completely different approaches of solving this problem. We have already stated that ROMIO uses the 2-Phase-Method for reading and writing data. Thus, the whole range of data including so-called holes is read rather than accessing only the data, which is actually needed. Since ViPIOS approach handles this problem by recursively accessing the data according to the file view (compare the chapter on the implementation of derived datatypes), i.e. no *unnecessary* data is accessed and therefore I/O overhead is decreased. Unfortunately, no test results are available yet to some internal inconsistencies of the system.

Chapter 8

Conclusion

In that thesis we presented ViMPIOS which is a portable MPI-IO implementation on ViPIOS. Besides giving basic information on MPI we explained the features of MPI-IO. Later we discussed the ViPIOS interface and the implementation process of ViMPIOS. The last section was dedicated to an extensible regression suite.

The advantage of ViMPIOS over other MPI-IO implementations is the client-server approach. Thus, the application programmer has the possibility to specify the number of client processes, i.e. the processes which handle the application program. What is more, the number of server processes can be specified explicitly. As a consequence, file I/O can be optimized in this way. However, the current implementation does not support file hints. We plan to implement that feature in the near future. Providing file hints allows the user to give information about the I/O nodes such that the ViPIOS server automatically chooses the I/O node which is best suited for the particular task.

Bibliography

[1] http://cesdis1.gsfc.nasa.gov/linux-web/beowulf.html.

[2] http://www.mcs.anl.gov/home/thakur/romio.

[3] http://www.mcs.anl.gov/mpi/mpich.

[4] Peter Brezany, Thomas A. Mueck, and Erich Schikuta. Language, compiler and parallel database support for I/O intensive applications. In *Proceedings of the International Conference on High Performance Computing and Networking*, volume 919 of *Lecture Notes in Computer Science*, pages 14–20, Milan, Italy, May 1995. Springer-Verlag. also available as Technical Report of the Inst. f. Software Technology and Parallel Systems, University of Vienna, TR95-8, 1995.

[5] Peter Brezany, Thomas A. Mueck, and Erich Schikuta. A software architecture for massively parallel input-output. In *Third International Workshop PARA'96 (Applied Parallel Computing - Industrial Computation and Optimization)*, volume 1186 of *Lecture Notes in Computer Science*, pages 85–96, Lyngby, Denmark, August 1996. Springer-Verlag. Also available as Technical Report of the Inst. f. Angewandte Informatik u. Informationssysteme, University of Vienna, TR 96202.

[6] The MPI-IO Committee. Mpi-io: A parallel file i/o interface for mpi, April 1996.

[7] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[8] Message Passing Interface Forum. Mpi: A message-passing interface standard, May 1994.

[9] Message Passing Interface Forum. Mpi-2: Extensions to the message-passing interface, July 1997.

[10] William Gropp Rajeev Thakur, Ewing Lusk. *User's Guide for ROMIO, a High-Performance, Portable MPI-IO Implementation.* Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, October 1997.

[11] Chance Reschke, Thomas Sterling, and Daniel Ridge. A design study of alternative network topologies for the beowulf parallel workstation. In *Proceedings, High Performance and Distributed Computing, 1996*, 1996.

[12] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings, International Conference on Parallel Computing, 1995*, volume 1, pages 11–14, August 1995.

[13] Heinz Stockinger. Dictionary on parallel i/o. Master's thesis, Department of Data Engineering, University of Vienna, February 1998.

[14] Rajeev Thakur, Alok Choudhary, Rajesh Bordawekar, Sachin More, and Sivaramakrishna Kuditipudi. Passion: Optimized I/O for parallel applications. *IEEE Computer*, 29(6):70–78, June 1996.

[15] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, October 1996.