Fast Parallel Non-Contiguous File Access*

Joachim Worringen, Jesper Larsson Träff, Hubert Ritzdorf

C&C Research Laboratories, NEC Europe Ltd.
Rathausallee 10, D-53757 Sankt Augustin, Germany
{worringen,traff,ritzdorf}@ccrl-nece.de
http://www.ccrl-nece.de

Abstract

Many applications of parallel I/O perform non-contiguous file accesses: instead of accessing a single (large) block of data in a file, a number of (smaller) blocks of data scattered throughout the file needs to be accessed in each logical I/O operation. However, only few file system interfaces directly support this kind of *non-contiguous* file access. In contrast, the most commonly used parallel programming interface, MPI, incorporates a flexible model of parallel I/O through its MPI-IO interface. With MPI-IO, arbitrary non-contiguous file accesses are supported in a uniform fashion by the use of derived MPI datatypes set up by the user to reflect the desired I/O pattern.

Despite a considerable amount of recent work in this area, current MPI-IO implementations suffer from low performance of such non-contiguous accesses when compared to the performance of the storage system for contiguous accesses. In this paper we analyze an important bottleneck in the efficient handling of non-contiguous access patterns in current implementations of MPI-IO. We present a new technique, termed *listless I/O*, that can be incorporated into MPI-IO implementations like the well-known ROMIO implementation, and completely eliminates this bottleneck. We have implemented the technique in MPI/SX, the MPI implementation for the NEC SX-series of parallel vector computers. Results with a synthetic benchmark and an application kernel show that *listless I/O* is able to increase the bandwidth for non-contiguous file access by sometimes more than a factor of 500 when compared to the traditional approach.

1 Introduction

MPI [4, 10] incorporates a flexible model of parallel I/O, called MPI-IO [4, Chapter 9], which allows MPI processes to read and write data from external files, either independently or collectively. MPI-IO supports much richer file access patterns and operations than for instance the standard POSIX I/O interface [6] which is available at the operating system level.

An essential component of the semantics of MPI-IO are the so-called derived or user-defined MPI datatypes [10, Chapter 3]. MPI derived datatypes can describe arbitrary placements of related data, specifically including non-contiguous layouts of data blocks. Figure 4 gives an example of

*SC'03, November 15-21, 2003, Phoenix, Arizona, USA.

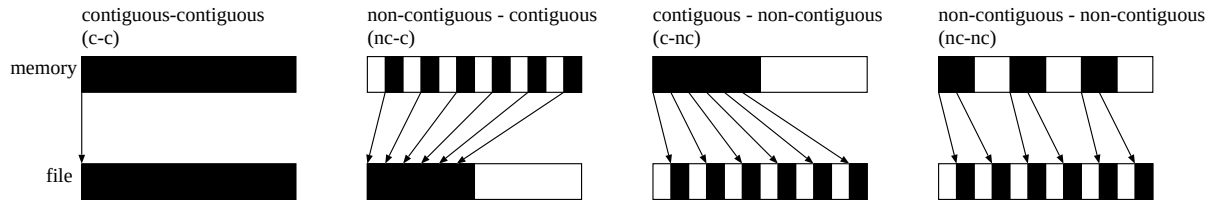

Figure 1: Accessing a file with MPI-IO: contiguous and non-contiguous fileviews and memory datatypes

such a non-contiguous datatype: contiguous blocks of length `blocklen` are placed in a vector, with a distance of `stride` between the beginning of each block. When `stride` is greater than `blocklen`, this datatype is non-contiguous.

By using MPI datatypes to describe the parts of a file visible to an MPI process, it is possible to specify arbitrarily complex, possibly non-contiguous access patterns. In this way the MPI datatype serves as a kind of *filter* between the MPI process and the file. Such a filter is called a (non-contiguous) *fileview*. Non-contiguous access patterns can be used to partition a file among a set of processes – each process sets up a different (non-contiguous) fileview and thus accesses different parts of the file even though all processes call the same read- or write-functions of MPI-IO with the same parameters. Figure 1 illustrates the four different combinations of layout of data in memory (the *user buffer*) and file:

- *contiguous – contiguous (c-c):* Moving data between a contiguous user buffer and a contiguous block of the file is the standard I/O application.
- *non-contiguous – contiguous (nc-c):* Moving data between a user buffer which is described by a non-contiguous datatype and a contiguous block of the file can be performed via a single intermediate buffer which contains the contiguous ("packed") representation of the non-contiguous data. Using the packed data, the file can be accessed as in the *c-c* case.
- *contiguous – non-contiguous (c-nc)*: Moving data between a contiguous user buffer and a file which is to be accessed via a non-contiguous fileview can be handled via a series of individual file accesses. Alternatively, *data sieving* [11] can be used: A large portion of the file is read into an intermediate memory buffer (the *file buffer*) using a single file access. Data is then copied between the user buffer and the file buffer (and written back to the file in the case of write accesses).
- *non-contiguous – non-contiguous (nc-nc)* is a combination of *nc-c* and *c-nc* and can be performed by applying the methods described above in sequence.

MPI-IO defines *collective* I/O operations in addition to the usual *independent* I/O operations. Collective operations must be called by all processes which have opened the file before they succeed. Collective I/O operations paves the way for optimizations to the actual file accesses that are not possible for independent I/O operations. For instance, the file can be partitioned among the processes to allow for a higher accumulated bandwidth. In this case data are exchanged (collectively) among the processes by MPI communication. Also data sieving is generally more efficient for collective I/O because of a better ratio of actually accessed data in the file buffer relative to the file buffer size.

There has been a fair amount of work on the efficient implementation of the MPI-IO model, including performance studies on different platforms and file systems, see e.g. [1, 2, 7, 9, 11, 12, 13, 17]. Most work is about the implementation of the well-known, portable open-source ROMIO implementation of MPI-IO [12]. This work, however, has dealt with the optimization of the necessary file accesses for non-contiguous fileviews. For independent I/O, techniques like data sieving are used [11, 13]. For collective I/O, data sieving is combined with techniques like *two-phase I/O* [11, 13]. Orthogonal work has dealt with hiding file access time through active buffering and I/O threads [2, 7], or optimizations for specific, low-level file systems [1, 9]. To the best of our knowledge, no attention *per se* has been paid to the efficient handling of non-contiguous, typed MPI data buffers as are always involved in non-contiguous file access. ROMIO for instance uses linear lists of $\langle offset, length \rangle$ tuples to describe the placement of individual, consecutive data blocks within a data buffer. For large-scale datatypes, as they are commonly used for the definition of fileviews, this naive approach which we term *list-based I/O* creates several performance problems as we will elaborate in Section 2.

In Section 3 we present a new technique, termed *listless I/O*, that completely avoids the use of *ol-lists* for handling non-contiguous file accesses. Instead, *listless I/O* employs a technique called *flattening-on-the-fly* to handle MPI datatypes [14]. The *listless I/O* implementation of MPI-IO has been integrated into MPI/SX [3], NEC's highly optimized MPI implementation for the SX-series of parallel vector computers. In Section 4 we evaluate the performance of the *listless I/O* MPI-IO implementation on NEC SX-6 and SX-7 parallel vector computers by comparing it with *list-based I/O* for a highly parameterizable, synthetic benchmark as well as for the application kernel BTIO from the NASPAR benchmark suite [16].

For the remainder of the paper, we use the term *filetype* for the MPI datatype which is associated with a file. Likewise, *memtype* shall denote the MPI datatype which describes the user buffer in memory. A *non-contiguous file* is a file which is accessed through a non-contiguous fileview.

2 Conventional Techniques for Non-Contiguous File Access

To understand the improvements that are achieved by *listless I/O*, it is helpful first to look at the overheads that occur when performing non-contiguous file access with *list-based I/O*.

The MPI-IO model defines *independent* and *collective* file access routines. The independent routines will succeed when called by any process, without coordination with other processes. In contrast, collective file access routines must be called by all processes which have opened the file before they can succeed at any process. This allows for optimizations inside of MPI-IO by coordinating the processes to create more efficient types of file accesses.

2.1 Flat Representation of MPI Datatypes

A straightforward representation of an MPI datatype is as an *ol-list* of $\langle offset, length \rangle$ tuples with each tuple describing a contiguous block of the datatype by its length and its offset relative to a lower bound. This *list-based* approach is used by ROMIO and, to the best of our knowledge, by all other MPI-IO implementations. It is simple to implement and use, but has a number of inherent, significant drawbacks:

Memory consumption: The explicit description of each contiguous block is completely opposite to more succinct, implicit descriptions of the data that are possible because of the repetition of patterns in most MPI datatypes. In particular, the type constructors of MPI lead naturally to a tree-like representation of user defined data types. The explicit representation leads to a

memory consumption for the representation which blows up linearly with the number N_{block} of disjoint elements in the datatype, even for simple (but frequently used) datatypes like vectors. As an extreme example, for blocklengths less than 16 bytes within a non-contiguous datatype, the list-based representation of a datatype consumes more memory than the actual data within the datatype.

Traversal time: Directly related to the size of the representation is the traversal time to gather the required information from it. For file accesses with a non-contiguous fileview, it is often necessary to determine the absolute position of the n-th element, or to perform an inverse search which means determining the sequence number n of the element at a given absolute file position. On average, because file accesses may be performed at arbitrary positions within a fileview, this means accesses to $N_{block}/2$ elements of the associated *ol-list*.

Copy time: The non-succinct datatype representation also affects the time needed to copy larger batches of non-contiguous data as each tuple requires a separate copy operation. This prevents using more efficient copy operations, like gather-scatter operations on systems which support such. Furthermore, using a *list-based* representation of derived MPI datatypes *each* read and write access to a contiguous block of data requires one more read access to the linear list to get the offset and length of this element. The negative performance impact of this requirement increases with reduced average length of the contiguous blocks.

When a new non-contiguous fileview is set up for the first time, ROMIO creates the *ol-list* for the associated datatype. This process is called (explicit) *flattening* of the datatype. The *ol-list* is then stored for re-use in case a new fileview is set up with the same datatype. If a non-contiguous memtype is used, another *ol-list* is created for this datatype for each access (these lists are not stored beyond the single access operation).

2.2 Independent List-Based File Access

A process accessing a non-contiguous file independently uses the *ol-lists* to find the offset in the file where the first byte is to be accessed. Although the fileview is described by a single MPI datatype, the file can be accessed with the granularity of the *elementary type (etype)*. The *etype* is another MPI datatype upon which the datatype of the fileview is built. As a consequence, the file can be accessed at offsets which are located inside a fileview. This is a different situation than packing or unpacking data with the `MPI_Pack` and `MPI_Unpack` routines which only handle entire datatypes [10, Section 3.12]. As a consequence, the determination of the absolute start- and end-offsets within the file requires linear traversal of the *ol-lists*.

Once the absolute file offsets are determined, the *data sieving* operation starts. The part of the file specified by the calculated offsets is read blockwise into the file buffer using contiguous read operations. In each iteration, data between the file buffer and the user buffer are exchanged, traversing the *ol-lists* for the filetype and, if necessary, the memtype. For write accesses, the file buffer is written back after each block. This requires that the related region of the file is locked to prevent non-related data from being overwritten by now obsolete data in the gaps in the file buffer.

2.3 Collective List-Based File Access

With the *two-phase method* [13, 13], a collective file access is performed by two groups of processes: *io-processes* (IOPs) that actually perform file accesses, and *access-processes* (APs) that do not do so, but instead communicate with the IOPs which perform the file access on their behalf. The

partitioning of the group of processes which opened the file into IOPs and APs is arbitrary, but usually either all processes function as IOP and AP at the same time, or a single process per SMP-node serves as IOP.

As the IOPs perform the file access on behalf of the APs, they need to know their fileviews which in the non-contiguous case usually differs among all processes. This means that every AP needs to build a specific *ol-list* for every IOP which accesses a part of the file for the AP. These lists are then sent to the IOPs. This many-to-many communication creates a substantial communication overhead as the size of these *ol-lists* may match or even exceed the size of the actual data requested. This is the case for a single-strided vector of doubles, where for each data element of eight bytes another 16 bytes need to be sent which describe the offset and length of the data element in the file buffer of the IOP[1]. Additionally, the number of elements N_{coll} in these *ol-lists* is not limited to the number of contiguous blocks in the non-contiguous filetype (N_{block}). Instead, they must contain a tuple for each contiguous block of data in the range of the file that the IOP will access, as seen through the fileview of the concerned AP. This means that N_{coll} is independent of N_{block}, but depends on the total extent of the file access.

The file accesses are performed blockwise; after each block is read into the file buffer by an IOP, it applies the data sieving technique on it for each AP that has delivered (write access) or requested (read access) data contained in this block. For this purpose, the IOP sets up an MPI indexed datatype from the relevant part of the *ol-list* received by each AP. This datatype is then used to communicate (*receive* for write accesses, *send* for read accesses) the actual data with the AP. After the file access for this block is complete, this datatype is freed again. It makes no sense to cache this datatype as it cannot be re-used.

For collective write operations into a non-contiguous file, ROMIO performs an optimization by determining if the combined accesses performed by all processes into a given range of the file result into a contiguous access. This is the case if *every* byte in this file range is modified by at least one write access. To determine this, each IOP merges the *ol-lists* of all processes for this file range (which it has received before). If this results in a single block, it is not necessary to read data for the current file range into the file buffer before copying the new data into the buffer (as is done by data sieving). Instead, it is valid to copy the new data directly into the non-initialized file buffer, which is then written back to disk. This optimization can significantly enhance the effective write bandwidth. However, the merging of the *ol-lists* into a single list can become time consuming as it scales with $O(\sum_{p=1}^{P} N_{block}(p))$.

2.4 Summary of Overheads

From the discussion above, we can identify the following overheads for list-based non-contiguous file access:

- Creation of the *ol-lists* for filetype and user buffer datatype takes time $O(N_{block})$.
- The storage of these lists may require significant amounts of memory, namely $N_{block} \cdot (\texttt{sizeof(MPI\_Aint)} + \texttt{sizeof(MPI\_Offset)})$ bytes.
- Navigating within the file (like positioning the file pointer of the file system according to the logical position within the fileview) requires traversal of on average $N_{block}/2$ list elements per access.

[1] This calculation assumes that 64-bits are used for offset and length.

- The creation and exchange of *ol-lists* for collective access is even more expensive than the initial flattening of the filetype. This is due to the necessity for an AP to create an *ol-list* for the complete portion of the file that an IOP will access on its behalf. The length S_{access} of this portion may be an arbitrary multiple of the extent $S_{extent(\text{ftype})}$ of the filetype. This leads to costs in time and memory of $O(S_{access}/S_{extent(\text{ftype})} \cdot N_{block})$ for each combination of an AP and an IOP[2].
- Copying the contiguous blocks of a non-contiguous datatype does not only require to copy the data itself, but also to read the corresponding $\langle offset, length \rangle$ tuple before the copy operation.

3 Improving Non-Contiguous File Access

To avoid the time- and memory-consuming usage of *ol-lists*, a different way to handle non-contiguous MPI datatypes is required. In the following we describe our new *listless I/O* approach, and how it was integrated into the existing ROMIO-based MPI-IO section of MPI/SX.

3.1 Flattening on the Fly

Instead of explicitly flattening MPI datatypes into *ol-lists* to be able to navigate through memory and file buffers, a method which provides for efficient, direct access to data blocks within a non-contiguous MPI data buffer is sought. The *flattening-on-the-fly* technique for efficient packing and unpacking of typed MPI buffers already used in MPI/SX provides this functionality [14]. In contrast to the generic, recursive algorithm used for instance in MPICH [5], *flattening-on-the-fly* is able to identify and copy large chunks of evenly spaced, non-contiguous data by gather-scatter operations as available on vector processors like the NEC SX-series. The actual packing and unpacking of data is performed in a non-recursive loop. Although originally designed to take advantage of the hardware support for gather-scatter operations, *flattening-on-the-fly* has been extended to be efficient on scalar architectures, too, by better exploiting data locality. This is borne out by the implementation of *listless-io* for PC architectures described in [17].

The *flattening-on-the-fly*-interface is very simple and consists of two internal functions (not accessible at the MPI user level). The function

```
MPIR_ff_pack(srcbuf, count, datatype, skipbytes, packbuf, packsize, copied)
```

packs (possibly) non-contiguous data from the user buffer `srcbuf` into a contiguous buffer `packbuf` of size `packsize`. The inverse function

```
MPIR_ff_unpack(packbuf, packsize, dstbuf, count, datatype, skipbytes, copied)
```

unpacks data from a contiguous `packbuf` into a non-contiguous user buffer `dstbuf`. In both cases, the non-contiguous data is described by an MPI `datatype` and a repetition `count`. This is similar to the internal interface for handling non-contiguous MPI datatypes as offered by MPICH. However, for both functions the number of bytes to be skipped, assuming a contiguous representation of the data, before starting the actual packing/unpacking must be specified. The ordering of the bytes in `packbuf` relative to `srcbuf` and `dstbuf` is determined by the *type-map* associated with the MPI datatype, see [10, Chapter 3]. Both functions return the number of bytes actually `copied`, which can be no larger than `packsize`. Thus, the *flattening-on-the-fly* interface can be used to pack or unpack parts of the complete data described by the MPI datatype and the repetition count. This

[2] S_{access} may be 0 for certain pairings.

is required if the contiguous `packbuf` buffer (which is usually used to transmit the data from one process to another) cannot be made large enough to contain the complete data.

Both pack and unpack functions of the *flattening-on-the-fly* interface are *efficient* in the sense that the time taken to pack and/or unpack typed, possibly non-contiguous data is directly proportional to the number of bytes to be packed/unpacked, plus a low-order term for the depth of the tree describing the MPI datatype (which is usually small). In particular, the number of recursive calls during pack/unpack is *independent* of any internal counts in the MPI datatype, and the pack/unpack time is *independent* of the number of `skipbytes` to be skipped in either source or destination buffer. Another crucial property of *flattening-on-the-fly* is that the actual copying of the data takes place outside the recursive traversal of the datatype. By this, it is possible to take advantage of gather-scatter operations typically available on vector machines.

3.2 Integration into ROMIO

The next step is to use these efficient functions for packing and unpacking non-contiguous data to replace the list-based packing and unpacking functionality in ROMIO.

The implementation needs to consider only the case where either filetype or memtype is non-contiguous. Considering the required packing and unpacking steps for file accesses with non-contiguous fileviews, it becomes clear that there are certain differences to the packing and unpacking that is performed when sending MPI messages of non-contiguous data.

3.2.1 Datatype Navigation

When sending non-contiguous messages, the access granularity is the datatype. For accessing a file, the access granularity is the *etype* from which the *fileview* is derived. Thus, file accesses may start or end *within* a datatype. This requires additional means to navigate within a datatype which are described below.

To position the file pointer inside the filetype it must be possible to determine the extent of a (virtual) data buffer for which a given number of bytes of MPI typed data has been written or read. Conversely, it must be possible to determine how many bytes of data are contained in an arbitrary typed buffer. This functionality is provided by two, new internal functions (see Figure 2).

The function

```
MPIR_Type_ff_extent(dtype, skipbytes, size)
```

returns the extent of a virtual typed buffer in the case that `size` bytes of data would have been unpacked from a contiguous buffer according to the datatype `dtype` after first skipping over `skipbytes` bytes.

On the other hand the function

```
MPIR_Type_ff_size(dtype, skipbytes, extent)
```

returns the amount of data that can be contained in a virtual buffer of datatype `dtype` with extent `extent` after skipping over `skipbytes` bytes. Using these functions, we can easily perform the calculations needed to toggle between absolute or filetype-relative positioning in the file, and determine the amount or extent of data that a buffer currently contains or is able to hold.

Both navigation functions are efficient in the same sense as the functions `MPIR_ff_pack` and `MPIR_ff_unpack`. The time to compute size and extent by these functions depends only on the depth of the tree representing the MPI datatype, and not on any internal counts or displacements herein.

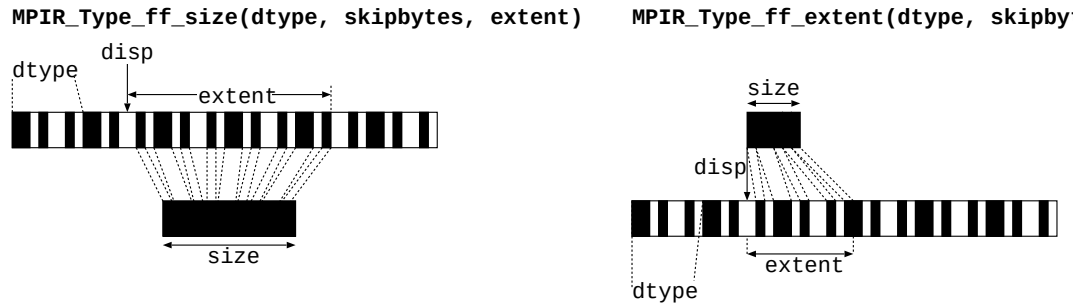

Figure 2: Datatype navigation using `MPIR_Type_ff_size` and `MPIR_Type_ff_extent`. The start displacement `disp` is found by skipping `skipbytes` of actual data from the start of the buffer.

3.2.2 Buffer Limit Handling

When packing or unpacking non-contiguous data for sending and receiving messages, it is always the internal communication buffer holding the packed data which is limited in size. The buffer which stores the unpacked representation of the data is the user buffer provided by the application which (per MPI definition) can hold the complete non-contiguous representation of the data. This is not true when accessing a file with a non-contiguous fileview: the file buffer can hold only a fraction of the data to be accessed, but has to do so using the non-contiguous representation of the data.

This problem of the lack of control of buffer limits applies for all accesses of the file buffer and is illustrated in Figure 3 for an *nc-nc*-style access. For a write operation, a fraction of the file is read into a file buffer of size f, and data from the non-contiguous representation of the data in memory is *packed* into an additional *pack buffer* of size p. From the pack buffer[3], it is *unpacked* into the file buffer using `MPIR_ff_unpack`. For message-passing communication, the amount of data to unpack is determined by the size of the `packbuf` buffer to unpack *from*. Here, the situation is different: the size of the file buffer to unpack *to* is the limiting factor. To be able to use `MPIR_ff_unpack` in this case, we need to determine the amount of data in the contiguous pack buffer that the destination buffer can hold in its non-contiguous representation using `MPIR_Type_ff_size`.

The need for the `MPIR_Type_ff_extent` function is a bit more subtle. When unpacking a block of data from the pack buffer into the file buffer, the `MPIR_ff_unpack` function would unpack data into a position offset from the beginning of the file buffer by the extent of the data before the current block. This is larger than zero except for the first block, and thus unpacking would in general result in data being written beyond the allocated file buffer. To cope with this problem, the address of the file buffer has to be adjusted to a virtual file buffer by subtracting the extent of the preceeding blocks. The size of the data written in previous blocks is known, and a call to `MPIR_Type_ff_extent` returns the extent needed for buffer adjustment.

For read operations, the direction of the data movement is inverted, and the problem described above occurs when packing data from the file buffer into the pack buffer.

3.2.3 Individual and Collective Access

Using *listless I/O*, the file accesses are still done in the same manner as with *list-based I/O* as we continue using the basic techniques of *data sieving* and *two-phase I/O* for non-contiguous file

[3]For an *c-nc*-style access this would be the user buffer.

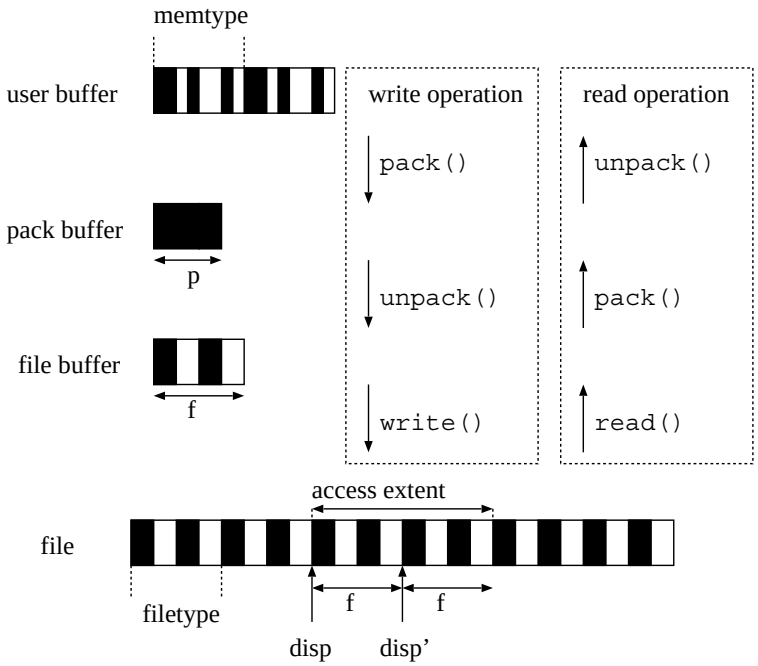

Figure 3: Utilization of intermediate buffers for non-contiguous memtypes or filetypes

access in our implementation. However, most of the related code was replaced with the *listless I/O* algorithms, eliminating all use of *ol-lists* in the generic I/O functions. Accordingly, *ol-lists* are no longer created[4].

Listless I/O achieves a very important improvement of the *two-phase I/O* technique concerning the handling of the fileviews of remote processes. In the list-based approach of ROMIO, *ol-lists* have to be exchanged for each collective access between APs and IOPs. In our approach (which we named *fileview caching*), a compact representation of each process' filetype and displacement are exchanged *only once* when the fileview is set up. For all collective accesses, each IOPs can perform the necessary file accesses with an AP's fileview using this cached information, without the need to exchange any information besides the actual file data. For the compact representation of MPI datatypes the same ADI as in the MPI/SX implementation of one-sided communication is used [15].

For collective write operations, an important optimization is to avoid filling the file buffer from the file before copying data into it. Using *listless I/O*, the approach of merging lists (as done with *list-based I/O*) is not feasible. Instead, we merge the complete fileviews of all processes into an *mergeview* at the same time a new fileview is established. This *mergeview* is made up of a `struct` datatype which contains all filetypes with identical displacements and suitable repetition counts (the *mergetype*). This approach is limited to cases where all processes use identical displacements in their fileviews. However, this is normally the case since the displacement is used to skip irrelevant portions of the file (like a header) which are the same for all processes. Using this *mergeview*, determining if the collective access for a given file range results in a contiguous access is simple:

[4]Some file systems like NFS and PVFS use their own file access functions for independent accesses and thus still require the list-based representation. In these cases, the *ol-lists* are still created, but not used in the generic access functions.

we simply call `MPIR_Type_ff_size` for the *mergetype*, with the displacement and offset arguments describing the file range that is to be accessed. If the returned size is equal to (or greater than) the given extent, the collective access will be contiguous. This assumption holds true because of the restrictions on datatypes that can be used as *etypes* and *filetypes* as specified in the MPI-IO standard: they must not have any negative displacements, and indexed and structured types must have monotonically increasing displacements [4, Chapter 7, Section 7.1]. Therefore, each byte may only be written once through each individual fileview, which is also true when creating the *mergeview* by overlaying these fileviews.

3.3 Summary of Eliminated Overheads

By *listless I/O* all overheads listed in Section 2.4 have been eliminated. In particular:

- No *ol-lists* have to be created or stored.
- There are therefore no costs for traversing *ol-lists*; all data type copying and navigation is done by the *flattening-on-the-fly* functions and the functions described in Section 3.2.1.
- Also no exchange of *ol-lists* for collective file accesses is necessary. Only compact representations of the fileviews are exchanged once.

4 Performance Comparison

In this section we present a first evaluation of our new approach by comparing it to the current list-based implementation in ROMIO. First, we perform a systematic analysis using an MPI *vector type* of the impacts of the relevant scaling parameters *vector length*, *vector blocksize* and *number of processes* on the bandwidth achieved. Next, we evaluate the performance improvement that *listless I/O* is able to achieve in an application scenario with the application kernel benchmark BTIO [16].

4.1 Systematic Analysis

For the systematic analysis, we designed a highly configurable benchmark which we term *noncontig*. It is a synthetic benchmark which employs a non-contiguous fileview based on a vector-like datatype

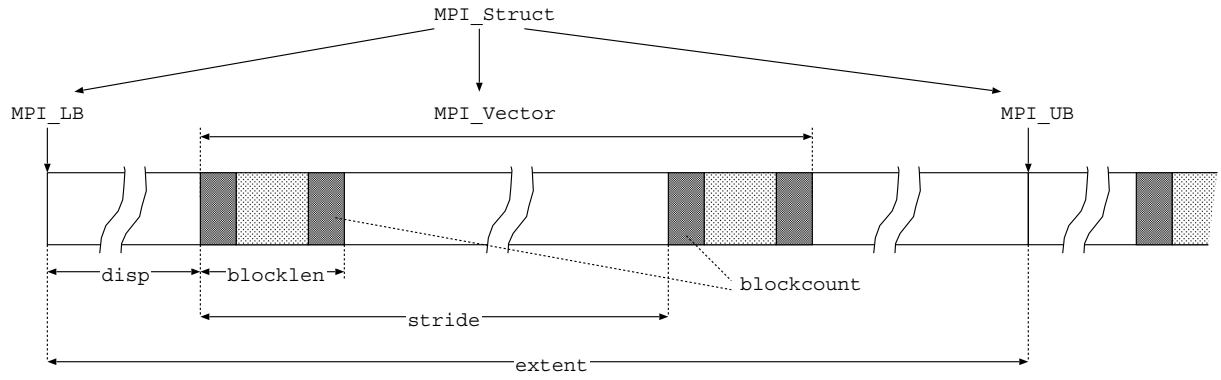

Figure 4: MPI datatype used for filetype and memtype in the *noncontig* benchmark

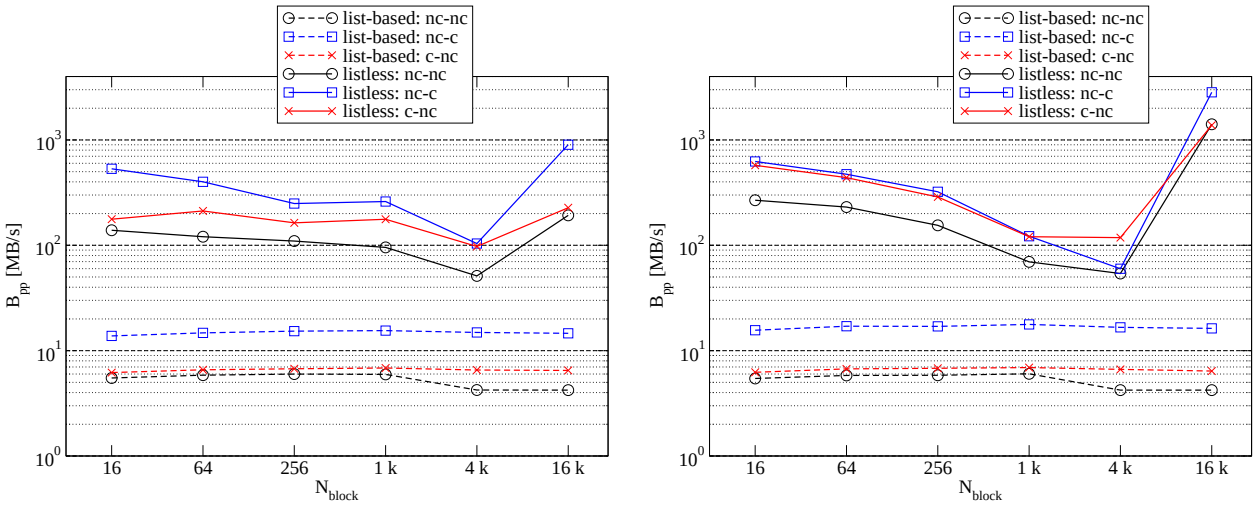

Figure 5: Scaling of I/O bandwidth for different vector lengths N_{block}: Bandwidth per process for **independent** write (left) and read (right) access (*noncontig* benchmark, $S_{block} = 8\,\text{byte}$, $P = 2$)

(see Figure 4). It writes and subsequently reads back a contiguous or non-contiguous memtype and measures the bandwidth at which those file-accesses are performed.

The most important test parameters are the number of elements in the datatype (`blockcount`) and the size of these elements (`blocklen`). Additionally, the size of the file can be specified. The test can perform independent or collective file accesses.

The datatype is different for each process p. For P processes, the displacement is $\texttt{disp} = p \cdot \texttt{blocklen}$, and the stride between the blocks is calculated as $\texttt{stride} = P \cdot \texttt{blocklen}$. This means that the file accesses of all processes are not overlapping.

For this test our test platform is a NEC SX-6 parallel vector computer. The benchmarks were run on a single, non-dedicated node which is equipped with 8 CPUs and 64 GB of RAM. The local file system that was used has a sustained bandwidth of about 6.5 GB/s for write access and 8 GB/s for read access.

We ran a number of test series with the *noncontig*-benchmark to evaluate the performance of *listless I/O* in comparison with *list-based I/O* and to study the scaling behavior of both approaches. For that purpose, we varied one parameter for each test series:

- *Vector length*: The vector length, described by the `blockcount` parameter N_{block}, will influence the time needed to build and traverse the *ol-lists*.
- *Vector blocksize*: The size S_{block} of the contiguous blocks in the vector (`blocklen`) determines the relative overhead for the individual copy operations and the creation and communication of *ol-lists* for collective operations using the *two-phase method*.
- *Number of processes*: As always, the performance achieved for different numbers P of processes performing a parallel task on the same platform indicates the scalability of the chosen algorithm. For collective file access, it determines the communication characteristics of the the *two-phase method*.

The bandwidth per process B_{pp} for an increasing vector length is shown in Figure 5 for independent access and in Figure 6 for collective access. In both cases, *list-based* access achieves a nearly

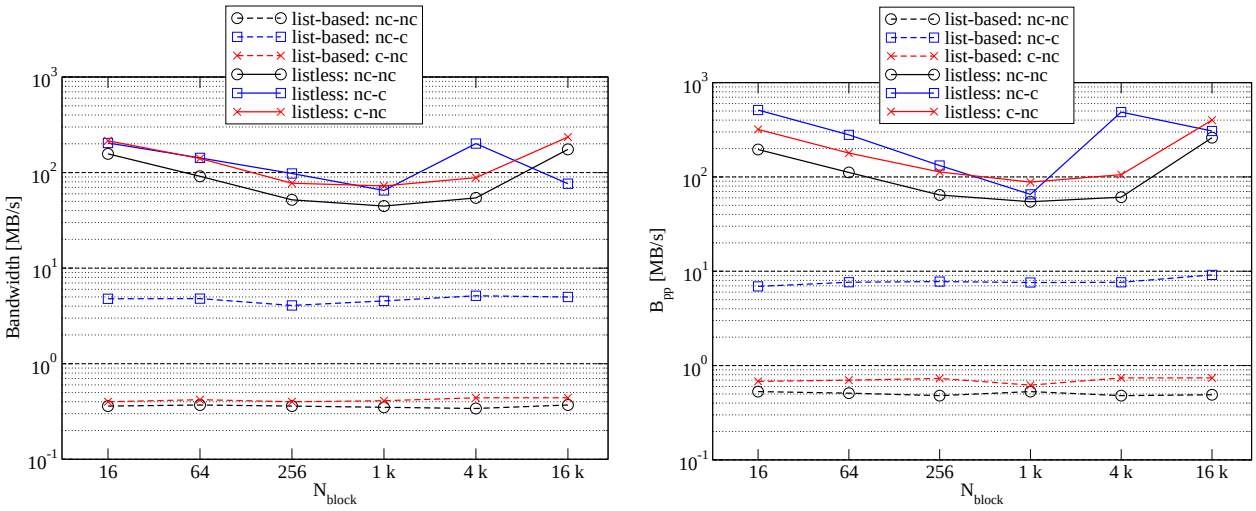

Figure 6: Scaling of I/O bandwidth for different vector lengths N_{block}: Bandwidth per process for **collective** write (left) and read (right) access (*noncontig* benchmark, $S_{block} = 8$ byte, $P = 8$)

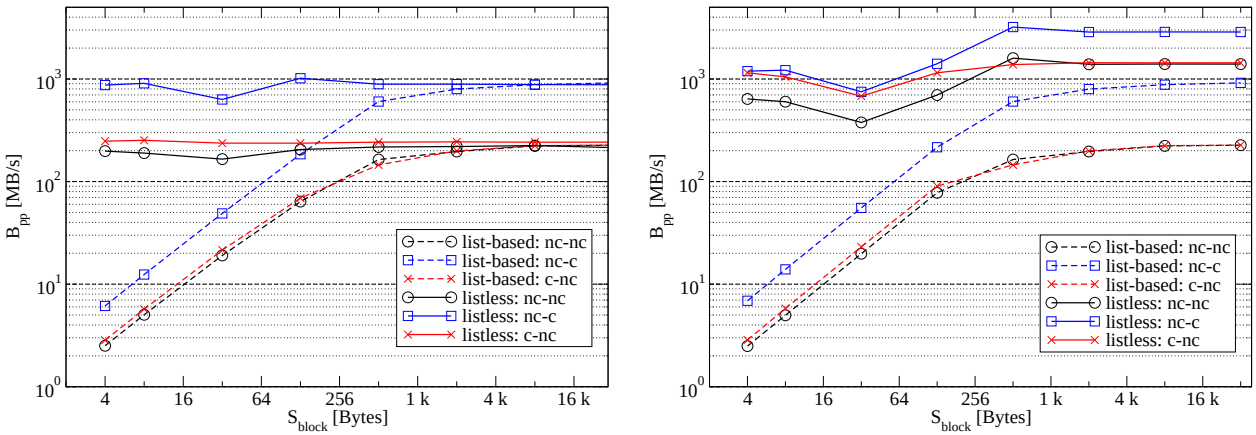

Figure 7: Scaling of I/O bandwidth for different vector blocksizes S_{block}: Bandwidth per process for **independent** write (left) and read (right) access (*noncontig* benchmark, $N_{block} = 8$, $P = 2$)

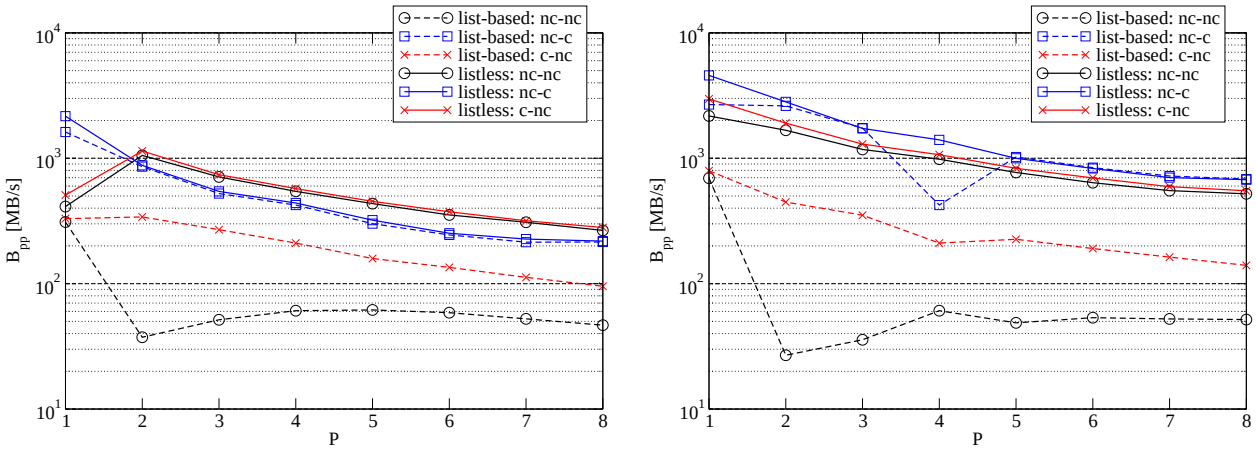

Figure 8: Scaling of I/O bandwidth for different process numbers P: Bandwidth per process for **collective** write (left) and read (right) access (*noncontig* benchmark, $16 < N_{block} < 128$, $S_{block} = 2048$ bytes)

constant bandwidth for all vector lengths. However, this bandwidth is on a very low level: For non-contiguous files (*c-nc* and *nc-nc*), independent accesses result in $B_{pp} < 10\,\mathrm{MB/s}$ and collective accesses never achieve more than 1 MB/s. Contiguous file accesses (*nc-c*) are three times faster for independent operations and more than 10 times faster for collective operations. This shows the constant amount of overhead caused by the exchange of the *ol-lists*. The overhead for the additional *ol-list*-traversal and copy operations for the *nc-nc*-access causes a performance decline of about 10%. Because the vector length only affects the time to locate a position in the file by traversing the *ol-list*, it does not have a significant impact on B_{pp} for this sequential access pattern.

In contrast, using *listless I/O* results in radically increased B_{pp} values. The improvement varies between a factor of 3.6 and 330 for independent accesses and between a factor of 8.6 and 540 for collective accesses. For independent accesses, this improvement is caused by using *flattening-on-the-fly*, which tries to make optimal use of the vector architecture of the NEC SX system and which eliminates all overheads associated with the *ol-lists*. Collective accesses are additionally accelerated by the *fileview caching* which avoids all additional communication apart from the necessary data transport. However, using *listless I/O* B_{pp} is not constant for all values of N_{block} as it is for *list-based I/O*. This is not caused by processing overheads inside of *listless I/O*, but is due to inherent characteristics of *flattening-on-the-fly*. When trying to find the optimal vector length for the required copy operations, there is a trade-off between the repetition counts found on different levels of the datatype. In the present case, the trade-off is between the repetition count of the datatype itself and N_{block} inside each datatype. This choice can possibly be optimized further.

Figure 7 shows B_{pp} for a fixed $N_{block} = 8$ and $S_{block} = 2^3, \ldots, 2^{14}$ bytes with $P = 2$ processes performing independent write and read accesses. As can be clearly seen, the performance advantage of *listless I/O* diminishes with increasing block size. This is due to the reduced number of copy operations performed by *list-based I/O* (for a fixed amount of data), which increases the efficiency of each copy operation. It is worth pointing out that *listless I/O* never performs worse than *list-based I/O*.

In Figure 8, we can observe that the ratio of the I/O performance between *listless I/O* and *list-based I/O* remains constant across most of the given range of processes. The performance of

list-based I/O degrades due to the required exchange of *ol-lists*. This does not apply to *listless I/O*. For *nc-c*-accesses, the performance of *listless I/O* and *list-based I/O* is nearly identical due to the large block size (as discussed above). The ratio for *c-nc*-accesses is about three for write and four for read accesses. For $P > 1$, *nc-nc*-accesses with *list-based I/O* suffer an additional performance decrease due to additional list-based data copy operations by the APs. This leads to a performance ratio of typically 8 for write and 10 for read accesses. The accumulated bandwidth of all processes remains nearly constant for all test cases which indicates saturation of the file system's performance for a single process.

4.2 Application Kernel Benchmark

For a more application-oriented evaluation of the performance effects of *listless I/O*, we chose the well-known BTIO benchmark [16], a variant of the BT benchmark of the NASPAR benchmark suite, also used in other studies of MPI-IO performance [13]. In addition to the computations performed by the BT benchmark, BTIO writes the complete array data to a file after a specified number of time steps. BTIO makes full use of the MPI facilities to handle multi-dimensional arrays:

- BTIO uses an MPI datatype to specify the non-contiguous distribution of each process' domain in memory. This datatype is created by calling `MPI_Type_create_subarray`.
- A similar datatype is used to specify the fileview of the output file (which is shared between all processes of the application).
- Based on these datatypes, BTIO is able to perform the complete I/O operation for the array with a single collective call to `MPI_File_write_at_all`.

Due to these implementation characteristics, BTIO is a good example of the advantage of assigning as much of an I/O task as possible to the MPI library. The version of BTIO used for our experiments is based on the vectorized code for the BT benchmark also used in [8].

BTIO can only be run with square number of processes and is preconfigured for different problem sizes (classes A through D). For the purpose of this comparison, it is sufficient to look only at the I/O performed by BTIO. The characteristics of the I/O operations can be described by the number N_{block} of disjunct data blocks written to file, the size S_{block} of these blocks. For each class and number of processes, BTIO uses a (nearly) constant value of S_{block}, therefore the resulting amount of data written out in each time step can be calculated as $D_{step} = P \cdot S_{block} \cdot N_{block}$. The resulting total amount of data written in a single run of BTIO can be calculated as $D_{run} = N_{step} \cdot D_{step}$. Per default, BTIO uses $N_{step} = 40$. Tables 1 and 2 show the related values for the different BTIO experiments that we performed.

The test platform for the BTIO experiments is a NEC SX-7 parallel vector computer which is similar to the SX-6 system, but with 32 instead of 8 CPUs per node. The increased number of CPUs comes along with an increased memory bandwidth. The benchmarks were run on a single, non-dedicated node which is equipped with 32 CPUs and 256 GB of RAM. The local file system

Class	Grid	D_{step}	D_{run}
B	$102 \times 102 \times 102$	42 MByte	1.7 GByte
C	$162 \times 162 \times 162$	170 MByte	6.8 GByte

Table 1: Characterization of BTIO's I/O data volume

Class	P	N_{block}	S_{block}
B	4	5202	2040
	9	3468	1360
	16	2601	1020
	25	2080	816
C	4	13122	3240
	9	8748	2160
	16	6561	1620
	25	5248	1296

Table 2: Characterization of BTIO's non-contiguous file access pattern (S_{block} is given in bytes)

Class	P	t_{no-io}	$\Delta t_{io-list-based}$	$\Delta t_{io-listless}$	r_{io}	$B_{io-list-based}$	$B_{io-listless}$
B	4	69.61	0.85	0.41	2.07	1976	4098
	9	47.46	1.32	0.76	1.74	1273	2211
	16	35.24	1.16	0.65	1.78	1448	2585
	25	30.45	1.57	1.47	1.07	1070	1143
C	4	190.44	4.31	2.29	1.88	1578	2969
	9	134.11	2.86	1.42	2.01	2378	4789
	16	97.63	2.42	1.27	1.91	2810	5354
	25	82.13	4.76	3.94	1.21	1429	1726

Table 3: Performance comparison between *list-based I/O* and *listless I/O* for different BTIO problem classes and process numbers P. Time values t are given in seconds; bandwidth values B in MBytes/s.

that was used has the same performance characteristics as the one used on the SX-6 system in the previous chapter.

We ran the classes B and C with the number of processes $P = \{4, 9, 16, 25\}$ processes. The results of these runs are given in Table 3. For each run, we give the following numbers:

- t_{no-io} is the execution time (in seconds) for the equivalent BT run without I/O.
- $\Delta t_{io-list-based}$ and $\Delta t_{io-listless}$ are the time differences for a BTIO run with *list-based I/O* or *listless I/O*, respectively.
- r_{io} is the ratio of I/O time using *list-based I/O* and *listless I/O*.
- $B_{io-list-based}$ and $B_{io-listless}$ are the effective I/O bandwidth numbers (in MB/s) achieved for the complete run by either *list-based I/O* or *listless I/O*.

The increase of I/O performance experienced by BTIO may seem lower than expected if looking at the results of the *noncontig* benchmark. However, *listless I/O* does in fact achieve an approximate two-fold increase in effective I/O bandwidth for this application-oriented test case (see values for r_{io}). To understand the difference in the behavior of *noncontig* and BTIO, it is necessary to take a closer look at the factors which influence the performance of collective non-contiguous I/O for the *list-based* and *listless* approach, respectively, and how they relate to each other for the BTIO benchmark:

- *Amount of data*: It is obvious that the more data an application writes or reads, the more important fast (parallel) I/O is for the total performance. The advantage of *listless I/O* will then be more visible in the total application execution time. However, for a real judgment of importance of I/O performance for the total application performance, it is necessary to put the amount of I/O in relation to the amount of computation and communication. This can be done by counting the number of times that each element (typically a floating point number) of the working set is moved between memory and file system, and the number of compute operations performed on it. At the same time, the processing latencies for an I/O and compute operation on a single element need to be put in relation to finally determine the I/O intensity of an application.

  This can be illustrated for an example taken from the shown BTIO runs. The class C problem being solved with 16 processes performs $n_{flop} = 2.8 \times 10^{12}$ double precision floating point operations, while $n_{ioop} = 0.85 \times 10^9$ double precision elements had to be written to disk. This gives a ratio $r_{comp-io} = \frac{n_{flop}}{n_{ioop}} > 3000$. If we consider an application with less than 10% of I/O time for the total execution time to be *not* I/O bound, then this test will not be I/O bound on a platform with an I/O subsystem which can process double precision floating point elements not less than 300 times slower than the CPU's are able to. Our test platform can (for this test) process 28.4×10^9 double precision floating point elements per second with its CPUs and 0.35×10^9 double precision floating point elements per second with its I/O system (using *list-based I/O*), resulting in $r_{comp-io} = 81$. This approximate calculation shows that the BTIO benchmark is not really I/O intensive on the tested platform. However, this is not a problem as long as we want to compare the performance of different I/O solutions as we do in this paper.

- *Filetype granularity*: The smaller the blocksize S_{block} of the non-contiguous fileview is, the greater the advantage of *listless I/O* over *list-based I/O* will be. This is quantified in Figure 7. We can also see in this figure that for $S_{block} > 1\,\text{KB}$, the packing via *ol-lists* becomes as fast as packing via *flattening-on-the-fly* for write access. As S_{block} is larger than 1 KB in most BTIO test cases (see Table 2), *listless I/O* can not gain an advantage this way. However, for collective file access, it reduces the overhead of the *ol-list* processing. For BTIO, this leads to a doubling of the file access performance.

- *Number of processes*: If more processes are used to collectively access a fixed amount of data (as it is the case for the scaling of P for a given BTIO problem size class), more but smaller messages are exchanged. This may lead to a performance decrease. On the other hand, accessing a file system in parallel may increase the accumulated bandwidth if the file system is using a storage system with a suitable striping configuration. The actual scaling property of a system thus depends on the relative scalability of the involved subsystems (see also next item).

- *File-system and memory performance*: With *list-based I/O*, there are three potential bottlenecks in the critical data path: the file system (reading and writing large blocks of data for data sieving), the interconnection network (for exchanging data and *ol-lists* during collective file access operations) and the local memory (for packing and unpacking using *ol-lists*, and the processing of *ol-lists*). Of these potential bottlenecks, *listless I/O* primarily addresses the local memory. Next to this, it reduces the communication load for collective file access. Still, file system performance is the limiting factor in many current high-performance computing systems.

5 Summary and Outlook

We have presented a new technique termed *listless I/O* to improve the performance of non-contiguous file accesses with MPI-IO by completely changing the handling of non-contiguous MPI datatypes within our MPI-IO implementation. Explicit lists of $\langle offset, length \rangle$ tuples are discarded, eliminating all overheads associated with this traditional way to represent non-contiguous MPI datatypes.

The performance evaluation shows that the actual performance gain depends heavily on the (non-contiguous) access pattern and the characteristics of the related subsystems of the platform. Bandwidth boosts of more than two orders of magnitude have been observed with the *noncontig* benchmark on a parallel high-performance vector system. For an application kernel, the bandwidth increase is limited to a factor of two. Independent of such variations, it is important to note that the presented *listless I/O* technique avoids severe performance degradations for highly non-contiguous access patterns which is crucial for providing high-performance systems with balanced performance characteristics. It could be shown that *listless I/O* is prepared to perform well for all cases of non-contiguous I/O. In a companion paper [17], we have shown that *listless I/O* is also effective on scalar cluster architectures, increasing the bandwidth about one order of magnitude.

The higher the bandwidth of the used file system is in relation to the bandwidth of the memory system and message passing interconnect, the more important *listless I/O* is to avoid a bottleneck for non-contiguous I/O. More detailed performance evaluations should be performed to further analyze the performance impacts of *listless I/O*. This performance analysis needs to include different file systems and different communication topologies. Especially the behavior in complex applications is of interest, as the tests with BTIO indicate. Applications sometimes use more complex filetypes like multi-dimensional arrays, which are accessed in different manners. This should give further insight into the performance behavior of *listless I/O*.

Optimizations of non-contiguous I/O are still possible at other levels of MPI-IO. A more general optimization, especially for independent access, is the decision on the trade-off between data sieving and multiple file accesses.

Acknowledgment

We would like to thank Rob van der Wijngaart (NASA Advanced Supercomputing Division) for supplying us with the vectorized version of the BT code used also in the comparison presented in [8].

References

[1] A. Ching, A. Choudhary, W.-K. Liao, R. Ross, and W. Gropp. Noncontiguous I/O through PVFS. In *International Conference on Cluster Computing*, pages 405–414, 2002.

[2] P. M. Dickens and R. Thakur. Evaluation of collective I/O implementations on parallel architectures. *Journal of Parallel and Distributed Computing*, 61(8):1052–1076, 2001.

[3] M. Gołebiewski, H. Ritzdorf, J. L. Träff, and F. Zimmermann. The MPI/SX implementation of MPI for NEC's SX-6 and other NEC platforms. *NEC Research & Development*, 44(1):69–74, 2003.

[4] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI – The Complete Reference*, volume 2, The MPI Extensions. MIT Press, 1998.

[5] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable imlementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996. See also `http://www-unix.mcs.anl.gov/mpi/mpich/`.

[6] D. Lewine. *POSIX Programmer's Guide*. O'Reilly and Associates, Inc., 1991.

[7] X. Ma, M. Winslett, J. Lee, and S. Yu. Improving MPI-IO output performance with active buffering plus threads. In *International Parallel and Distributed Processing Symposium (IPDPS)*, page 68, 2003.

[8] L. Oliker, J. Carter, J. Shalf, D. Skinner, S. Ethier, R. Biswas, J. Djomehri, and R. van der Wijngaart. Evaluation of cache-based superscalar and cacheless vector architectures for scientific computations. In *Supercomputing*, 2003. `http://www.sc-conference.org/sc2003/`.

[9] J.-P. Prost, R. Treumann, B. Jia, R. Hedges, and A. Koniges. MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS. In *Supercomputing*, 2001. `http://www.sc2001.org/papers/pap.pap186.pdf`.

[10] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998.

[11] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, 1999.

[12] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, pages 23–32, 1999.

[13] R. Thakur, W. Gropp, and E. Lusk. Optimizing noncontiguous accesses in MPI-IO. *Parallel Computing*, 28:83–105, 2002.

[14] J. L. Träff, R. Hempel, H. Ritzdorf, and F. Zimmermann. Flattening on the fly: efficient handling of MPI derived datatypes. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users' Group Meeting*, volume 1697 of *Lecture Notes in Computer Science*, pages 109–116, 1999.

[15] J. L. Träff, H. Ritzdorf, and R. Hempel. The implementation of MPI-2 one-sided communication for the NEC SX-5. In *Supercomputing*, 2000. `http://www.sc2000.org/proceedings/techpapr/index.htm\#01`.

[16] P. Wong and R. F. V. der Wijngaart. NAS parallel benchmarks I/O version 2.4. Technical Report NAS-03-002, NASA Advanced Supercomputing (NAS) Division, 2003. See `http://www.nas.nasa.gov`.

[17] J. Worringen, J. L. Träff, and H. Ritzdorf. Improving generic non-contiguous file access for MPI-IO. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 10th European PVM/MPI Users' Group Meeting*, volume 2840 of *Lecture Notes in Computer Science*, 2003. To appear.