be pre-allocated to already contain all information; thus avoiding additional communication to data servers during file creation. PVFS supports some bulk operations: when listing a directory, for example, a large number of directory entries are transferred in one response. Also, PVFS incorporates read-only client side caches for metadata.

- **Consistency semantics:** The semantics of a file system define how the API calls are translated to file system manipulations and accesses. Of special interest is how and when a server (and a client) realizes concurrent modifications made to file system objects. Relaxed guarantees open the potential for additional optimizations. For instance, when a client does not need to realize all modifications made in the past, it could save communication that would refresh the state of cached objects. Also, operations could be deferred on a client to bundle them with future requests, then other clients would not realize modifications until they were communicated to the servers.

  Assume a client which tries to create a number of files; this client could request file creation of every file individually, or it could avoid some of the communications by assuming the state of the directory is still valid. With strong consistency semantics, locking is mandatory to perform bulk operations because with a lock modifications of the file system status could be prevented. For example, Lustre allows to issue a set of metadata operations at once by locking metadata, and Lustre pre-creates files, too.

  With relaxed consistency semantics, modifications will be lost, if client or server crash prior to execution and thus persistence is not ensured. However, in this case, locking mechanisms might become costly, too; the crash has to be detected and the lock must be released to continue operation.

  Consistency semantics are defined by the application program interface. POSIX for example has very restrictive semantics since I/O operations have to be applied in the same order as issued, even if they are non-overlapping. Thus, communication with the I/O subsystem must be serialized to guarantee proper handling. This is a major drawback of the POSIX interface. Most programs could be adapted to loose semantics to exploit parallelism. PVFS does not provide guarantees for I/O data in case of concurrent operations, data could be a mixture of data blocks of different requests. However, with PVFS all metadata operations are ordered in a specific way to guarantee metadata consistency of objects in the accessible namespace[27].

  Further information on concurrent access is given when MPI-IO semantics are discussed (see Page 49).

- **Locking strategy:** Locking as a mechanism can be used to prevent concurrent access to one (or multiple) file system objects. Depending on the required file system semantics, measures must be taken to ensure that clients access the current state of the file system. While GPFS and Lustre provide a locking protocol, PVFS does not offer locking; hence, when using PVFS the application must avoid concurrent access to file regions. In presence of a locking mechanism, the granularity of the lock defines the potential concurrency with related data and metadata – a lock could be valid for a file region, a logical file itself or whole directories. Different lock types could be available – depending on the type of the lock, another access type could be permitted. A particular lock, for instance, could prevent modifications to a file but allow concurrent reads, while another one may restrict the type of access possible. Locking algorithms can become expensive in a distributed environment, since they require a consistent view.

2.3. Message Passing Interface

The Message Passing Interface [Mes09] is a standard to enable inter-process communication. With MPI the user explicitly encodes the sending and receiving of data in the source code as function calls to MPI.

[27] There could be broken objects not yet linked to the namespace and inaccessible, though. These broken objects can be identified and cleaned by performing a file system check.

In this section the focus is placed on semantical aspects and communication performance implications imposed by the interface; features are only listed if they are relevant to this thesis.

The standard offers a rich set of *point-to-point* and *collective* functions to address many use-cases while keeping communication efficiency in mind. In point-to-point communication two processes communicate, one sends a message and the other receives it. With collective operations an arbitrary number of processes can participate to exchange data. In general, collective operations can be grouped into three classes depending on the number of processes that provide the initial data and the number of receiving processes: all-to-one (for example *MPI_Gather()* and *MPI_Reduce()*), one-to-all (e.g., *MPI_Bcast()*), all-to-all (e.g., *MPI_Allreduce()*).

With MPI-2 a set of functions to manipulate files are standardized. Directory operations are provided to a limited extent, only creation and deletion of files is supported. Basically, MPI-2 stretches communication concepts of MPI to I/O. I/O functions rely on communication between MPI processes with the I/O servers; in the context of I/O the MPI processes performing an I/O call are referred to as *clients*. MPI defines individual and collective I/O operations and permits non-contiguous access to file data.

First, Section 2.3.1 describes the rules how transferred messages are matched by a receiver. Then some MPI functions are introduced in Section 2.3.2. An excerpt of semantical aspects is provided in Section 2.3.3. Based on the semantical aspects, available optimization potential is elaborately discussed in Section 2.3.4. In Section 2.3.5 related work is presented to gear the MPI run-time towards system and application.

2.3.1. Matching Sends to Receives

A program can perform multiple collective calls and point-to-point operations at the same time. To ensure proper operation, data must be received by the intended receiver and not by an arbitrary process. Therefore, several rules are defined by MPI that match send messages to receives – the rules were chosen to permit efficient implementations but still offer a comfortable programming interface. One basic rule, for instance, is that messages are usually received in the order they are sent, i.e., whenever two messages match all rules of the receiver, then the one received first is given preference over the others.

Point-to-point communication Messages include metadata in an envelope which contains the *source*, *destination*, *tag* and *communicator*. A communicator describes a group of processes that participate and a context. The source and the destination are the *rank* of the respective processes in the specified communicator, i.e., their numeric ID within it. The tag is a numeric identifier that distinguishes messages from a sender, typically a tag has a user-specific semantic. For example, a message with Tag 1 could mean that this message contains input data, while Tag 2 indicates that computation is completed.

According to the parameters supplied to the receive operation, incoming message envelopes must match some or all of the information to be received. Wildcards can be used to receive a message from any rank of the communicator, or one with an arbitrary tag. However, the *communicator* must always match as specified by the MPI standard:

> *"A communicator specifies the communication context for a communication operation. Each communication context provides a separate "communication universe:" messages are always received within the context they were sent, and messages sent in different contexts do not interfere."* [Mes09]

Further, in a message envelope information about the transferred data types is contained. This information is not used to match sends to receives, but it helps the library to check the correct data transmission: When a message matches the conditions, received data is copied from the message into the output buffer on the receiver side. During this process the elementary datatypes specified on the sender side and receiver side are compared to ensure consistency. When they do not match the received data is invalid. Also, the number of elements to receive must match the number of elements transferred. While it is not possible to receive more elements than specified, because usually the provided buffer does not suffice, it is possible to receive

less data than expected. For example, a sender can send 10 integers while the receiver specifies a buffer for 50 integers; the actual number of elements received can be checked in the program after the receive completed.

Derived datatypes in the MPI standard allow developers to directly transmit non-contiguous regions in memory – that means multiple non-overlapping memory areas can be sent with one MPI call. Non-contiguous communication avoids the need to pack data explicitly in a contiguous buffer. MPI defines a large number of datatypes to ease definition of vectors, structures or expansions of primitive datatypes. By referring to already defined datatypes the user can compose arbitrarily nested datatypes. In the communication process the datatypes are converted to the receiver's machine architecture if necessary, and copied from/to the specified memory regions. Datatypes on sender and receiver side can be different, although the elementary datatypes below must match (it does not make sense to transfer an integer and receive a double value).

MPI-2 applies this concept to enable non-contiguous I/O (see Section 2.3.3 for details on how to use it). See section *Type Matching Rules* in [Mes09] for more details.

Collective calls MPI forbids interference of messages from point-to-point operations with messages sent by collective operations. To quote the standard:

> *"Collective communication calls may use the same communicators as point-to-point communication; MPI guarantees that messages generated on behalf of collective communication calls will not be confused with messages generated by point-to-point communication."* [Mes09]

Interference between collectives shall not happen as well, as it makes no sense for one collective operation to receive a message from another.

Different datatypes on sender and receiver are permitted, but the amount of data must match, according to the standard:

> *"The type-matching conditions for the collective operations are more strict than the corresponding conditions between sender and receiver in point-to-point. Namely, for collective operations, the amount of data sent must exactly match the amount of data specified by the receiver. Different type maps (the layout in memory [...]) between sender and receiver are still allowed."* [Mes09]

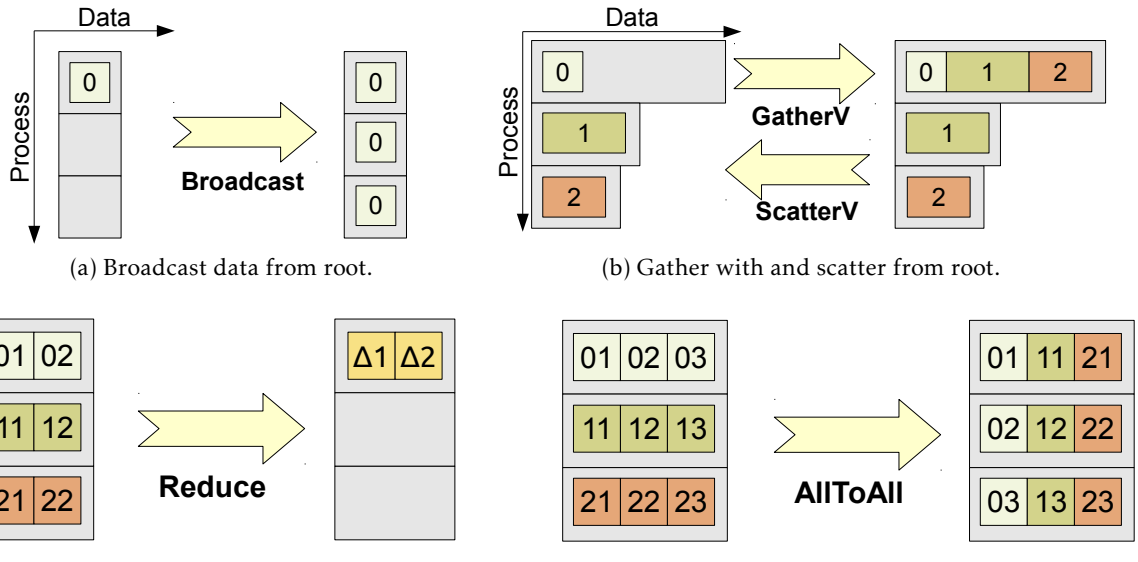
Consequently, MPI collective calls receive a message from another collective call iff the communicator is identical and the collective call matches. When one process of a communicator starts one collective call and another process starts a different call, then a deadlock occurs. MPI-3 will allow asynchronous collective calls; it is likely that multiple non-blocking starts of the same collective operation will be serialized to avoid false matching of messages.

2.3.2. Exemplary Collective Operations and Semantics

To foster discussion of performance aspects, the parameters and the invocation semantics for at least one representative of each collective pattern is listed. Data exchanged between processes is visualized in Figure 2.8. Gray boxes symbolize buffer space for input and output buffers – for the sake of simplicity it is not distinguished between input and output buffer. In the figure it is assumed, that *one-to-all* or *all-to-one* operations are performed with Rank 0 as root/target. The definitions and semantics are taken from [Mes09].

Synchronization behavior Once a collective function returns, the required operation for the local process has been performed in its output buffer; this implies that collective functions are blocking[28]. Collective operations do not imply synchronization. Synchronization between processes depends on the implementation of the specified semantics; for example, with a broadcast operation all processes must wait for

[28] With MPI-3 non-blocking collective calls will be introduced.



(a) Broadcast data from root.

(b) Gather with and scatter from root.

(c) Reduce data of multiple elements in root. Δi is computed from p_i (for all processes p in the communicator).

(d) All ranks exchange data.

Figure 2.8.: Data transfer scheme of a few collective operations.

the root process to be ready. But theoretically other processes could complete even if some processes did not call the collective operation, yet.

To quote a lengthy, but informative general introduction to collective calls:

> *"Collective routine calls can (but are not required to) return as soon as their participation in the collective communication is complete. The completion of a call indicates that the caller is now free to modify locations in the communication buffer. It does not indicate that other processes in the group have completed or even started the operation (unless otherwise implied by the description of the operation). Thus, a collective communication call may, or may not, have the effect of synchronizing all calling processes. This statement excludes, of course, the barrier function.*
> [...]
> ***Advice to users.***
> *It is dangerous to rely on synchronization side-effects of the collective operations for program correctness. For example, even though a particular implementation may provide a broadcast routine with a side-effect of synchronization, the standard does not require this, and a program that relies on this will not be portable.*
>
> *On the other hand, a correct, portable program must tolerate a synchronizing side-effect of a collective call. Thus, though it cannot be relied on the synchronization effect of a collective call, one must formulate the program to deal with it."* [Mes09]

Several functions can replace existing data in the input buffer directly with new data[29], that means input data is replaced with the result of the collective call. This enables the MPI library to conserve about half of the memory. Internally, the implementation might provide additional data buffers. However, buffer capacity should be minimized as the description of *All-to-All Scatter/Gather* in [Mes09] states: *"Users may opt to use the "in place" option in order to conserve memory. Quality MPI implementations should thus, strive to minimize system buffering."*

[29] To do so the user has to set the buffer to `MPI_IN_PLACE`.

Broadcast The `MPI_Bcast()` operation transfers data from a *root* process to all other processes in the group (Figure 2.8a), upon return of a callee the data is available in its local output buffer. The *signature*[30] of the call is:

```
MPI_BCAST( buffer, count, datatype, root, comm )
  INOUT   buffer     starting address of the buffer (choice)
  IN      count      number of entries in buffer (non-negative integer)
  IN      datatype   data type of buffer (handle)
  IN      root       rank of broadcast root (integer)
  IN      comm       communicator (handle)
```

The datatype can be an elementary datatype or a derived datatype.

Gather A gather operation transfers data of a fixed size from each process's buffer to the root buffer (Figure 2.8b). Data of the processes is stored subsequently in the buffer of the root, that is data of Process i is stored right in front of data from Process $i+1$. `MPI_GatherV()` customizes the amount of data sent by each process to the root.

Signature of the call:

```
MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype,
            root, comm)
IN  sendbuf     starting address of send buffer (choice)
IN  sendcount   number of elements in send buffer (non-negative integer)
IN  sendtype    data type of send buffer elements (handle)
OUT recvbuf     address of receive buffer (choice, significant only at root)
IN  recvcounts  non-negative integer array (of length group size) containing the
                number of elements that are received from each process
                (significant only at root)
IN  displs      integer array (of length group size). Entry i specifies the
                displacement relative to recvbuf at which to place the incoming
                data from process i (significant only at root)
IN  recvtype    data type of recv buffer elements (significant only at root)
                (handle)
IN  root        rank of receiving process (integer)
IN  comm        communicator (handle)
```

Gather can be thought of to be the inverse function to an scatter operation, which distributes subsequent blocks of data from the *root* among all processes. Broadcast is quite similar to scatter. Consider that Rank 0 transmits individual data to each process, which is indicated on the right side of Figure 2.8b. After the operation completed each process has its part of the data locally (left side of the figure).

Another explanation to the execution of collective operations is to specify the operation in terms of point-to-point operations – the definition for `MPI_GatherV()` is:

> *"[...] the outcome is as if each process, including the root process, sends a message to the root,*
> ***MPI_Send**(sendbuf, sendcount, sendtype, root, ...),*
> *and the root executes n receives,*
> ***MPI_Recv**(recvbuf + displs[j] · extend(recvtype), recvcounts[j], recvtype, j, ...)*
> *The data received from process j is placed into recvbuf of the root process beginning at offset displs[j] elements (in terms of the recvtype)."* [Mes09]

By setting the *displs* vector accordingly it is possible to change the order in which the data is stored, i.e., it is possible to store data from Process 2 at the beginning of the output buffer. However, it is erroneous to receive data from multiple processes in the same memory location.

[30] In the signature the description of the input and output parameters is given in a language independent notation according to the standard. Parameters are prefixed with the type (input, output or both).

Reduce Aim of a global reduction operation is to perform an associative operation like sum or minimum, across data provided by the processors of a group. A vector of *count* elements can be supplied in each process, then a reduction is performed for each element of the vector individually. In Figure 2.8c, each process has two elements, after the reduce the root process has the two independent results ($\Delta 1$ and $\Delta 2$) computed with the associative operation.

Signature of the call:

```
MPI_REDUCE( sendbuf, recvbuf, count, datatype, op, root, comm )
  IN  sendbuf   address of send buffer (choice)
  OUT recvbuf   address of receive buffer (choice, significant only at root)
  IN  count     number of elements in send buffer (non-negative integer)
  IN  datatype  data type of elements of send buffer (handle)
  IN  op        reduce operation (handle)
  IN  root      rank of root process (integer)
  IN  comm      communicator (handle)
```

The user can define further operations, which might also operate on derived datatypes – pre-defined operations work only on elementary datatypes. To quote [Mes09] regarding the semantics of the operation:

> *"The operation op is always assumed to be associative. All predefined operations are also assumed to be commutative. Users may define operations that are assumed to be associative, but not commutative. The "canonical" evaluation order of a reduction is determined by the ranks of the processes in the group. However, the implementation can take advantage of associativity, or associativity and commutativity in order to change the order of evaluation. This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating point addition.* ***Advice to implementors.***
>
> *It is strongly recommended that MPI_REDUCE be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of processors."*

The advice to the implementors impacts the performance, as numerically unstable floating point operations must be executed in the same order. Therefore, independent of the process mapping, the same operations must be executed, thus locality of the processes cannot be exploited in all cases. `MPI_Allreduce()` is a variant of reduce in which the result is stored on all processors – not only on the root process.

All-To-All With this operation each process transfers individual data to each other process; the amount of data exchanged between all processes is the same, see Figure 2.8d.

Signature of the call:

```
MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
  IN  sendbuf    starting address of send buffer (choice)
  IN  sendcount  number of elements sent to each process (non-negative integer)
  IN  sendtype   data type of send buffer elements (handle)
  OUT recvbuf    address of receive buffer (choice)
  IN  recvcount  number of elements received from any process (non-negative integer
    →)
  IN  recvtype   data type of receive buffer elements (handle)
  IN  comm       communicator (handle)
```

In this all-to-all function the user may specify that the data in one buffer is used as input and output, thus it can be replaced with new data. Effectively, this halves memory consumption.

2.3.3. Excerpt of MPI-IO Functions and Semantics

The I/O interface of MPI can be thought of as an extension of POSIX I/O for parallel programs. While the basic idea of file manipulation is connected to POSIX, the interface and semantics enable coding of portable and efficient parallel applications.

There are several MPI functions that deal with input and output of data. The semantics differ in the way the file pointer is used – either all processes share a common file pointer, or each process has its own file pointer. An orthogonal aspect is whether the call is blocking or non-blocking (and shall be executed concurrently with further program activity). Further, each process can perform I/O independently, or multiple processes can participate during the I/O (so-called *collective I/O*). As the functions are quite similar, only a subset of the available functions is introduced to enable later performance discussion: opening of a file, to enable non-contiguous file access by defining a file view, the non-blocking read operation and the collective write operation.

At last, the access semantics for multiple processes accessing the same file are discussed.

Opening of files With the collective function `MPI_File_open()` all processes of the communicator open a file. Once the file is opened individual (independent) or collective I/O can be performed by referencing the returned file handle. It is expected that all processes of the communicator participate in subsequent collective I/O operations with that file handle. The implementation must ensure that communication routines do not interfere with I/O operations. Therefore, no message from communication routines shall be received by an I/O operation and vice versa (see also Section2.3.1).

Signature of the call:

```
MPI_FILE_OPEN(comm, filename, amode, info, fh)
  IN comm       communicator (handle)
  IN filename   name of file to open (string)
  IN amode      file access mode (integer)
  IN info       info object (handle)
  OUT fh        new file handle (handle)
```

The specified filename on all processes must reference the same logical file – for example it is invalid for multiple processes to collectively open files on a temporary directory that cannot be accessed from all processes. The implementation defines how the filename is interpreted, ROMIO uses prefixes to distinguish various file systems. To access files in the PVFS namespace, for instance, the prefix "PVFS2://" is specified.

The file access mode is similar to the POSIX open call, with `MPI_MODE_CREATE` the call creates a new file and thus it alters the namespace of the file system if the file does not yet exist. Besides a delete function, there is no other way to alter metadata and the namespace with MPI functions.

Users can supply optimization parameters, which are in the form of key/value pairs, to the info object. Those parameters are referred to as *hints*. The `access_style` is an example of a reserved hint, the user can specify the temporal and spatial access patterns in which file access is performed until the file is closed – possible values are `read_once`, `read_mostly`, `write_once`, `write_mostly`, `sequential`, `reverse_sequential`, and `random`. Multiple access patterns can be defined, e.g., to announce that a file is read and written exactly once. Note that file hints can be changed with the `MPI_File_set_info()` function at run-time. Further information about hints is given in Section 2.3.5.

Defining a file view A file view defines how and which areas of the file are accessible by a process. This supports a better interpretation to the raw bytes of the file – by setting the data types contained, the implementation could perform file type conversion automatically. Further, non-contiguous file accesses are

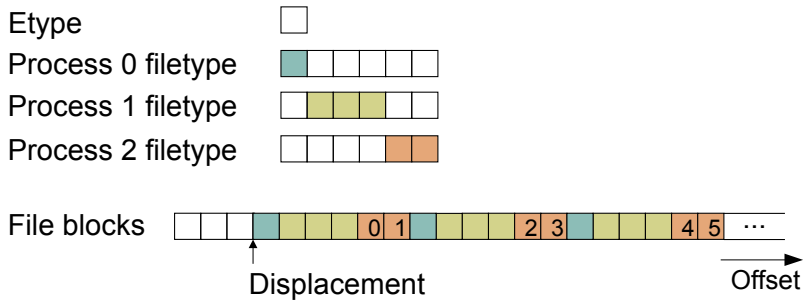


Figure 2.9.: Example partitioning of a file by using a file view. The accessible area for each process is colored, the offsets for accessible etypes are printed for Process 2.

implemented with the help of file views. First, the file is partitioned into regions which can be accessed, then non-accessible regions are skipped by subsequent I/O calls.

This function is collective – all processes which opened the file set their own view. While the file is open the file view can be altered by invoking `MPI_File_set_view()` again. It is invalid to change the view of a file while non-blocking I/O is performed.

Signature of the call:

```
MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)
  INOUT fh       file handle (handle)
  IN disp        displacement (integer)
  IN etype       elementary datatype (handle)
  IN filetype    filetype (handle)
  IN datarep     data representation (string)
  IN info        info object (handle)
```

The *elementary datatype* defines the unit of positioning and data access. From the program's point of view access to parts of an *etype* does not make sense, e.g., to read half of an integer. Only full etypes can be read/written and data positioning is relative to this datatype, i.e., the first etype is accessed, then the second etype and so on. For I/O calls the datatypes are restricted to use non-negative and monotonically increasing displacements.

The *filetype* sets how etypes form the whole file. Unimportant file regions are skipped, those non-accessible areas are usually referred to as *holes*. By using datatypes with holes non-contiguous file access is possible.

In *datarep* the interpretation of data types is set. Either the system's native representation is used – that means no conversion will be performed, or datatypes can be converted to a fixed representation. There are two alternatives that guarantees portability of the files created: An MPI implementation specific representation, which is portable across architectures as long as the same MPI implementation is used. And a standardized data type representation that is understood by all MPI implementations, but potentially slower.

An illustrated example of a collective file view for three processes is given in Figure 2.9. In this example the etype could be any portable datatype starting from a single integer to a complex record. Process 0 accesses the first etype, Process 1 the next three etypes and Process 2 the two following etypes. The datatypes are repeated infinitely in the file. In the figure the offsets for Process 2 are printed into the data blocks.

Independent non-blocking read With independent I/O, each process performs I/O without knowledge of the other processes' state. The operations are marked with an "*I*" in their name, read and write calls are treated similarly.

In the following example, the signature of the read call is provided:

```
MPI_FILE_IREAD(fh, buf, count, datatype, request)
  INOUT fh        file handle (handle)
  OUT   buf       initial address of buffer (choice)
  IN    count     number of elements in buffer (integer)
  IN    datatype  datatype of each buffer element (handle)
  OUT   request   request object (handle)
```

Accessed data is read from/written into the provided buffer with the specified memory datatype. If a file view has been applied like in Figure 2.9, then a process accesses only its visible data, e.g., if Process 2 reads 6 elements, then all 6 etypes of the example file are read in one call – thus, non-contiguous file access is performed. A potential data conversion is performed on the fly, too.

An independent non-blocking I/O operation just extends the signature of the blocking call by an opaque request object[31]. This opaque request object can be used to check the completion of the operation by MPI-1 functions such as `MPI_Wait()`.

Collective write In collective operations all processes which open a file might work together to access file data; this enables certain client-sided optimizations.

From the signature, a collective call is differentiated from the independent calls only by its prefix "`_all`". Depending on the implementation collective I/O operations might be synchronizing or not; the author of this thesis, however, has not seen non-synchronizing collective I/O in an MPI implementation, yet.

Signature of the call:

```
MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)
  INOUT fh        file handle (handle)
  IN buf          initial address of buffer (choice)
  IN count        number of elements in buffer (integer)
  IN datatype     datatype of each buffer element (handle)
  OUT status      status object (Status)
```

Access semantics Compared to the consistency model of POSIX, the MPI standard follows relaxed I/O semantics.

In the POSIX I/O semantics, each read and write operation is an atomic operation – concurrent operations are serialized and follow the *strict consistency* model. When a call finishes all manipulations become immediately visible – the first executed call is executed first and has to terminate before the next one can start. Consequently, when a `write()` and `read()` call are performed concurrently, then one of those two operations is scheduled first. Partly written data is not returned in the read function – if the read finishes first, then the old data is read, if the write finishes first, then read will retrieve all new data. Concurrently scheduled processes obey these semantics as well, ultimately this requires to serialize all file operations.

Sequential consistency relaxes the strict model slightly: The order in which the calls are invoked (in terms of wall-clock time) can now vary from the actual execution order. However, all processors have the same picture of the execution order, and from the perspective of a processor its operations are executed in the same sequence as they have been issued. These semantics are a consequence of the distributed nature of the system; it is hard to tell which operation started first. Since all processors see the same execution order, the operations are serialized, but not necessarily the first call issued is executed first.

Three consistency levels are available in MPI: sequential consistency of one file handle, sequential consistency with atomic mode and explicit, user-imposed, consistency.

[31] Non-blocking MPI communication routines such as `MPI_Isend()` use a similar technique.

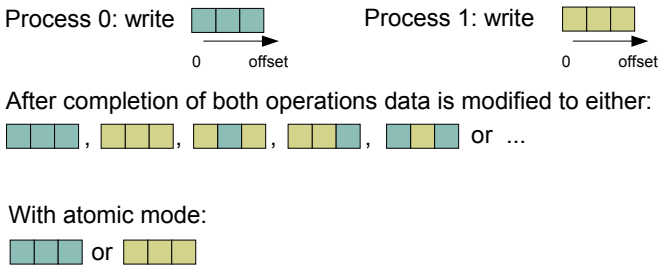


Figure 2.10.: Example of concurrent file operations, two processes write the same file regions.

In terms of MPI, the sequential consistency model implies that the behavior of accesses performed on one file handle follows any of the potential serializations of the execution, as implied by the synchronizations of the parallel program's processes: Read operations enforced to occur later in program execution than a write operation to the same file handle read the altered data, and writes to the same file regions overwrite previously written data. To illustrate this issue, consider two processes opening one shared file, one process writes some data, then both processes invoke a `MPI_Barrier()`[32] before the second process reads the data back. With the barrier, the sequential consistency enforces that the first process reads the same file region which was written before because in any execution the read always follows the write operation. Without the barrier, some old data (if the file existed before it was opened) or a mixture of already modified and old data could be returned to the other process.

This example illustrates that while each I/O operation by itself produces a well defined result, concurrent I/O to overlapping file areas result in unpredictable data – parallel accesses to overlapping file regions are not handled atomically as in POSIX.

Assume two processes opening one file, then each process writes three elements of local data. In theory the observed data in the file can be any combination of the data stored by the two processes, e.g., the first two elements from the first process and the remaining datatype from the other process – this example is illustrated in Figure 2.10. This kind of mixed data might break integrity of file records. Not only the datatypes but even the bytes in the file could be interleaved in the same fashion. In practice, file systems usually update data in the granularity of a full block. Thus, data of two processes is not mixed within one block but across block boundaries.

Atomic mode changes the behavior by serializing independent I/O to the same file handle if necessary, concurrent modifications by third-party processes or by using another file handle to access the same file are still under responsibility of the programmer (and user). If supported by the MPI library and the file system, atomic mode can be set by calling the collective function `MPI_File_set_atomicity()` with the file handle.

The sequential consistency of MPI does not specify the state of the file in regards to concurrent modifications by other programs – MPI might be unaware of these modifications. Also, the atomic mode does not protect the integrity of records in case multiple file handles point to the same logical file – it is undefined what data is accessed by independent processes and file handles. To deal with this issues the collective function `MPI_File_sync()` enforces to persist and to synchronize manipulations of one file handle with the file system. After a synchronization is performed, subsequent read operations access data updated by third-party software (or by the same program that used other file handles) in the background. Both opening and closing of a file imply this kind of synchronization.

The MPI semantics are quite comparable to the NFS semantics, as in NFS there is no cache consistency between different clients and the server. In MPI the set of processes which collectively open a file share the access with the sequential consistency model, this is exactly the behavior NFS enforces for one client

[32] A barrier is a synchronization point for all processes of the communicator. The processes wait until all of them reach an `MPI_Barrier()`, then they continue.

as each node uses local cache to increase performance.

MPI can share a single file pointer among multiple processes. Whenever a process reads or writes data, the file pointer is modified accordingly. The current semantics of a shared file pointer require a strict serialization of concurrent calls to avoid that two processes access the same file regions. However, the semantics define that the file pointer could be updated immediately after an I/O call is initiated. Thus, theoretically, the actual data transfer can happen concurrently.

2.3.4. Optimization Potential Within MPI

Several issues mentioned in Section 2.2 are related to performance of communication and I/O. The semantics of *Message Passing Interface* enable additional concepts that improve performance, some of them have been integrated on purpose into the interface. The potential that resides in using MPI is now discussed on an abstract level before details are provided on the state of the art.

In general, the usage of MPI within the application limits how well an MPI library could optimize interprocess communication – in the best case, the programmer uses the highest-level function to perform his tasks and ensures a balanced startup of all communication partners, i.e., all processes start synchronously. To reduce the complexity of the discussion, let us assume the hardware setup and the communication pattern of the application are fixed.

The performance of inter-process communication certainly depends on the hardware characteristics – the specific hardware configuration limits potential network throughput, computation power and available memory bandwidth. Therefore, it is our goal to exploit the available hardware resources to reduce application run-time.

There are two orthogonal concepts that have a large impact on performance: Either the process mapping to the existing hardware can be adjusted, or the MPI internal algorithms are adapted. Actually, many minor optimizations exist that may gear an implementation towards a system and a specific (application's) workload. For I/O calls all these options permit to optimize throughput, but even further optimizations can be applied to improve the access pattern and to decrease the burden on the I/O servers. A discussion of the state of the art that addresses the optimization potential mentioned is provided in Section 2.3.5.

Process mapping The logical MPI processes must be mapped to the available *processing elements*[33]. Without modifying MPI internally, *placing the MPI processes* in an appropriate manner on the processors yields a high potential to optimize communication and I/O behavior. However, selecting an appropriate mapping requires that the access patterns of the application are known before the program is executed.

Consider the example in Figure 1.8 (page 12). Assume, the 4 processes should be placed on two nodes with dual core processors (the example in Figure 1.6, page 9)). There are $4 \cdot 3/2$ options to map the 4 processes to the two nodes – the two processes on the first node are either $(0,1),(0,2),(0,3),(1,2),(1,3)$ or $(2,3)$. Actually, if the two nodes are equivalent, then the mapping $(0,2)$ is equal to $(1,3)$, the same with the mappings $(0,1)$ and $(2,3)$. In the example only neighboring processes communicate via point-to-point communication, but additionally a broadcast operation is performed. It can be seen that the mapping $(0,1)$ (or $(2,3)$) requires less inter-node communication than a mapping like $(0,2)$ because this way only one message is sent between process 2 and 1 over the slower network. The other two send operations can be performed within a single node. Therefore, this placement is typically faster.

Coming back to the difficulty of the process mapping, even under homogeneous hardware the number of possible permutations of the process mappings grows exponentially with the number of processes. It has been shown that the mapping problem is NP-hard. See [LB98, CCH+06] for a discussion of issues related to task scheduling. Tools have been developed that apply heuristics to place processes on the system topology.

[33] The mapping from processes to hardware is introduced on Page 36.

Communication algorithm Under the assumption that the placement is fixed, i.e., either because it is optimal or pre-defined, then there is potential to adjust the MPI internal algorithm towards the given application and network topology. To cope with the complex interplay of system and application, many different algorithms must be implemented in each MPI library. However, choosing the right algorithm for a given function call remains difficult. Presumably, suboptimal performance can be observed on every system (for at least some applications).

Point-to-point operations are of simple nature, as just two processes participate; therefore, the low-level network driver has a major impact on the performance. From the algorithmic point of view an MPI implementation has only a few options if it sticks to the standard's semantics. The semantics of `MPI_Send()` permit a completion of the operation, even if the receiver did not yet get the message; this allows the implementation to buffer the message either on the side of the sender or the receiver. Many variants of the point-to-point communication exist which enforce a certain buffering or synchronization behavior.

Collective operations are of much more interest. Related work shows the impact of the arrival pattern on performance and that tuning for individual message size is important [FPY07]. From a library's point of view, optimization can be done based on the parameters provided by the programmer. Typically, this includes the *memory datatype*, the *communicator*, *target/source rank* (for all-to-one or one-to-all operations), and the actual *amount of data* shipped with the call:

- The memory datatype is relevant because contiguous datatypes can be transferred via RDMA. Datatype on sender and receiver side can be be different, requiring to pack or unpack data.
- The communicator defines the participating processes in a given communication, even if an algorithm is optimal for one defined group of processes, for another set of processes a different algorithm might be faster. The problem of choosing the appropriate algorithm for a given group of processes is linked closely to network topology.
- The target rank is relevant because the implementation must perform the last operations on the target rank, therefore, for instance a spanning tree algorithm should put its root on the target rank. Similarly, for a one-to-all operation the source rank provides some information which must be transferred to all other clients.
- Depending on the amount of data the communication time varies. A basic consideration is that latency becomes more important for small messages, while larger messages are dominated by the bandwidth between the communication partners.

Concepts to optimize I/O Due to the complexity of I/O, that involves communication as well as server activity, there exists a rich variety of opportunities to optimize access patterns and metadata operations. First, a few hardware considerations are recapitulated from section 2.2.2. Those help in assessing the mentioned optimizations. From the perspective of the servers it is important that clients access as much data as possible in one I/O request, because it keeps the pipelines on the server busy and, furthermore, it enables sophisticated server-sided optimizations. A server can access data on block-level in a granularity of full blocks – modification of a few bytes of a block requires reading the full block – then modify it and write it back. Hence, performance is wasted. Due to the characteristics of hard disk drives, random access achieves only a small fraction of available performance (at least on hard disk drives) – the mentioned optimizations are all designed having disks as persistent storage in mind. Processing of a request requires some CPU and memory resources. As requests must travel over a network from client to server, processing involves additional delay.

To tackle the issues mentioned, the non-contiguous operations and collective calls have been defined in MPI-IO. In [TGL02] access to data with MPI is classified into *four levels of access*. The levels are characterized by two orthogonal aspects: contiguous vs. non-contiguous data access, and independent vs. collective calls. Thus, the levels are: contiguous independent, contiguous collective, non-contiguous independent and non-contiguous collective. Depending on the level a different set of optimizations can be thought of.

When using collective I/O calls, clients can collaborate to access data on I/O servers efficiently. Therefore, the clients exchange information about their spacial access pattern, then MPI orchestrates the I/O operations. Thus, optimizations are possible with collective I/O which are impossible to perform with individual I/O. For example, the MPI implementation might aggregate requests for neighboring blocks into a few larger request. This not only reduces the amount of requests, but also the resulting requests have a more sequential access pattern, than the individual operations. Also, the schedule for the requests can be adjusted. For example, the order in which data is requested can be swapped.

Further, to overlap computation or communication with I/O, analogous to non-blocking communication the non-blocking I/O is defined in the standard. A non-contiguous read or write operation usually triggers a sequence of contiguous data accesses within MPI. Internally, MPI could start the requests in another order, or perform a single larger access to avoid spamming the server with a high number of operations.

Our discussion for the access semantics of MPI on Page 49 revealed that its sequential consistency requires an implementation to persist file modifications by calling `MPI_File_sync()`. The access semantics can be exploited further to cache data in faster devices of the memory-hierarchy, to aggregate accesses, or to keep the file's data in an suitable internal data structure. However, until function invocation all data access could be cached in a buffer specific to the particular file handle.

Another possibility for the user is to provide additional information to MPI about the access pattern, or to change internal defaults to values which prove to improve performance. The implementation can honor this information to perform additional and better internal operations. Technically, in MPI-2 an opaque *info object* is defined, which basically consists of key-value pairs; this info object can be supplied to some MPI-IO functions to change their internal behavior. Note that the implementation is not forced to support or even obey hints; but the implementation can use it to (hopefully) tune the internals. Unfortunately, the hint is not incorporated in the function prototypes of collective calls.

2.3.5. State of the Art

Optimized collective calls With the upcoming of MPI many algorithms were proposed to optimize collective communication [KHB+99, TRG05]. Depending on the hardware, especially network topology and interconnect, the algorithm which achieves best performance varies.

Proprietary systems could add special hardware to realize or just support collective operations. The MPI library could be adjusted to utilize the hardware. On the SciCortex machine, for example, the network devices forward network packets for broadcast operations directly to their neighbors[34]. The new Mellanox Infiniband adapter enables offloading of communication operations. This way, the communication protocol can be handled directly on the network adapter [KST+11].

As algorithms must be handcrafted towards the system, for instance for a BlueGene [AHA+05, MK05], one major problem is to pick the best algorithm for a system. Heterogeneous systems complicate this issue, inhomogeneous hardware within a cluster makes it a tedious task to determine the best algorithms because the algorithm depends on the nodes that participate in the communication. Another issue is the interconnection of homogeneous clusters in a WAN or Grid environment. Potentially different MPI implementations on each homogeneous part of such a system must collaborate in an efficient way [BTR03, ITKT00].

The influence of the starting times of processes invoking a collective operation is measured and discussed in [FPY07]. In their work process synchronization and the pattern of process arrival is proven to make an important contribution to the efficiency of selected collective algorithms and it is claimed that application developers cannot control the arrival pattern.

[34] Unfortunately, SciCortex is out of business.

Choosing the best collective algorithm Several approaches have been developed that assist in determining the best algorithm and MPI configuration. The search for the best algorithm and its configuration is an optimization problem. Theoretically, all available implementations (and configurations) could be evaluated by *brute-force*. However, due to the large number of potential configurations often *historic knowledge* about previous execution is used to predict performance. To reduce the parameter space promising candidates could be identified by a *performance model*, or by classifying configurations manually. For example, the topology of a 3D-torus restricts the number of algorithms that could achieve best performance.

The *Abstract Data and Communication Library (ADCL)* [GH07] uses historic knowledge during the application run. ADCL assumes a program performs operations iteratively – in the first few iterations ADCL evaluates a set of MPI functions to determine which one is best suited for the given problem, then this function is applied to subsequent invocations. Evaluation is performed on the fly during the program run, every time a function is invoked another internal algorithm is evaluated, until the winning function is decided. Therefore, no artificial data is communicated to evaluate the performance of the functions. Compared to other solutions ADCL provides a new interface with higher abstraction than MPI, for instance, to allow users to specify neighbor communication explicitly. Communication is performed internally by calling `MPI_IRecv()`, `MPI_Recv()`, `MPI_Put` and other point-to-point calls. The function which performs best for a problem is called for subsequent invocations. Astonishingly, already on the abstract level of MPI function calls, the function best suited depends on application, problem size and system [GFBR10]. ADCL does not care about MPI-internals, it just treats MPI functions as a black box.

In [FG08], ADCL is extended to use historic knowledge across program executions. Therefore, achieved performance is recorded in XML files and used to predict performance of future executions. This reduces the number of high-level MPI functions to evaluate. Especially, functions which are well known for their slow execution on similar problems could be skipped, speeding up the run-time decision process of ADCL and consequently reduce application run-time.

Compared to ADCL, the *Self-Tuned Adaptive Routines for MPI Collective Operations (STAR-MPI)* provides a rich set of MPI implementations for collective operations by itself [FYL06], for instance a set of 13 algorithms is supplied for `MPI_Alltoall()`. Those algorithms are inspired by the algorithmic research on efficient collective communication. Similar to ADCL, STAR-MPI evaluates the performance of the implementations on the fly during program execution. In the first stage, during each collective function call STAR-MPI evaluates the performance of an alternative implementation. Once all implementations are tested multiple times, the processes pick the best function and use it for subsequent calls. While the winning function is used to realize the MPI call, the time spent for subsequent calls to this function call is monitored in order to deal with changes in workload and environmental conditions. The algorithm is adaptive because it tries to keep the current selection as long as possible but still adjusts to changes. For instance, if the difference between previous run-times is too high, then the second best algorithm will be chosen for further calls. STAR-MPI selection of algorithms depends also on the message size, for each size a best algorithm is kept. To reduce the training phase, algorithms which are expected to behave similarly are partitioned into groups, e.g., a group for inter-node communication, or for a switched network. Now, instead of checking all algorithms of all groups, the group is selected by testing only one algorithm of each group and choosing the best group for further evaluation. On average between two to three algorithms are grouped together.

Selecting the appropriate methodology for benchmarking and to interpret measured performance results is tedious and thus there are tools which assist in creating new benchmarks. *MPIBlib* [LRO08] is a library which bundles common features of MPI benchmarking suites including timing, repetition of experiments and calculation of statistics. Features of such a library can be embedded directly into the application or into the tools selecting the fastest communication implementation.

Process placement and topology awareness Besides tuning MPI internals, the mapping of the processes to the available CPUs and nodes plays an important role – as the application communication behavior should match the node-internal and external network topology to minimize communication time

(see Page 51). The *Moab* scheduler knows the physical layout of the cluster, which enables it to provide a mapping with low communication costs [CCS+06]. Mapping of processes to processors is done in *Hypertool* by a heuristic taking the communication traffic into account [WG90].

An automatic profile-guided approach is introduced in [CCH+06]. The *MPI Process Placement toolset* provides a set of tools: one tool records the communication profile, another one explores the system topology, at last an optimization algorithm determines a viable solution for the application mapping. This solution can then be applied in subsequent application executions by the user itself.

Neither of those mapping tools takes process I/O to servers into account, although at first glance it could be handled similarly to communication.

When a placement is chosen, the topology must be explored by MPI to utilize it. Starting with the recent Open MPI 1.3, *Carto*, a framework which determines the topology of the inter-process communication partners, is incorporated. By using *Carto*, for example a graph is built for the shared memory interconnect and one for the remote network. For each interconnect, the topology is stored in a weighted graph and available for other internal frameworks to tune their interconnection to remote processes. As an example, Open MPI can use the NIC closest to a particular sender process, or shared memory collective operations can take the memory distribution into account.

MVAPICH2 also aims to become topology aware to tune collective operations according to the topology. Recent work in this area has demonstrated the potential of this approach [KSVP10], for `MPI_Gather()` and `MPI_Scatter()` up to 50% improvement could be achieved on an Infiniband cluster.

Individual I/O Client-side I/O optimizations for MPI are presented in [TGL99]. *Data-sieving* is introduced to optimize independent non-contiguous I/O – it operates by issuing one request to access required data together with intermediate *holes*; unwanted data is just discarded. This avoids unnecessary seeking on hard disk drives and can improve performance, especially for very small blocks. However, it causes unneeded data to be read and transported to the clients.

General write-behind for MPI files is realized in [LCCW05, LCC+07], in their implementation all clients dedicate a piece of their memory to form a cooperative cache.

Non-blocking I/O enables overlapping communication (and computation) with I/O, the efficiency is analyzed theoretically and empirically in [BKL09]. Theoretically, a perfect overlapping of I/O with computation halves execution time.

To reduce the number of messages for non-contiguous I/O, Ching et al.[CCC+03] extend ROMIO to use ListIO in PVFS client-server communication.

In [KRVP07] a method to optimize non-contiguous (random) writes is introduced. Basically all write operations are appended to a logfile in the order they are performed. Consequently, non-contiguous operations are converted into sequential operations. This manipulation does not preserve the logical order of data in the physical file because the file format is different. However, when the file is read from the same file handle the actual file format must be recreated by replaying the log. Once a file is closed, it could be converted to the correct logical file and thus the sequential consistency of MPI is satisfied. Hence, this optimization has a good use-case for write-mostly access patterns, e.g., for checkpoints.

With shared file pointers, the actual data transfer could happen concurrently, however, in most cases I/O is suboptimal. Most implementations are not performing well because file system constructs such as a file lock or hidden files are used to realize the semantics of a shared file pointer. Some purely MPI-based implementations have been investigated and evaluated [LRT07].

Collective I/O The PVFS implementation of ROMIO reduces the metadata overhead of opening a new file by delegating the file open (and the potential create operation) to Rank 0 of the communicator specified

during the open call. All other participating processes receive the information required to access the file by a broadcast from the root and thus the burden of the metadata server is reduced.

The collective optimization of the *Two-Phase* protocol is discussed in [CCC+03]. With Two-Phase, processes exchange their spatial access pattern and coordinate amongst themselves, which process will access a sequential file domain – data to be accessed is partitioned among all clients. Then, (for a read operation) the clients repeat two phases: a set of the processes reads the assigned file domains sequentially, then during the communication phase data is shipped to the clients which needs it[35].

Multiple-Phase Collective I/O, an extended version of the Two-Phase protocol, is presented in [SIC+07, SIPC09]. With Multi-Phase I/O, the communication phase is split up in several steps, in which pairs of clients communicate with each other in parallel. These multiple steps are used to progressively increase the locality of the data to be accessed by aggregating more operations into larger blocks. Then, during the I/O phase, sequential blocks can be accessed; the block size of the sequential access depends on the number of steps performed.

In [HYC05], the collective I/O scheme is adjusted to exploit the features of Infiniband.

A cooperate cache is integrated in [PTH+01], this allows GPFS to cache read and defer write operations. With a write-behind strategy, write operations are buffered. Effectively, data updates and storing the changed data on the file system happens concurrently. To ensure consistency, every physical data block is assigned to exactly one client which performs all I/O operations.

In [Wor06] an adaptive approach is introduced which automatically sets hints for collective I/O, based on the access pattern, topology and the characteristics of the underlying file system.

Higher-level I/O optimizations The following two approaches are not optimizing MPI-IO by itself, instead they provide a layer above MPI that enables further interesting optimizations.

Initially, *SIONlib* [FWP09] was developed to deal with I/O forwarding and communication topology of a BlueGene system. This library channels I/O operations to a logical file from POSIX via MPI, i.e., in a program regular POSIX (e.g., `fwrite()`) calls are used, which are mapped to a set of shared files by using the MPI-IO interface. Depending on the underlying system, one or multiple files are generated to optimize performance. Conflicts on parallel access of a shared file are reduced, yet, the number of files is less than the number of processes which minimizes metadata overhead on the underlying file system. For instance, on a BlueGene, all processors routing to a particular I/O aggregator can be mapped to one physical file. Hence, the I/O forwarder does not have to share access to the file with other aggregators. The mapping from logical to physical files is hidden behind the library and transparent to the application. The user just specifies the number of files in the library open call and whether or not collective calls should be performed.

The *Adaptable IO System* (ADIOS) [LKK+08, LZKS09] provides an abstract I/O API and library that decouples application logic from the actual I/O setting. In an XML file, the desired I/O method can be selected and parameterized – I/O operation can be realized either with HDF5, MPI (collective or independent), POSIX or several asynchronous staging methods. With ADIOS, each I/O performed in a C (or Fortran) program is annotated with a name that can be referred to in the XML file. The amount of data accessed, datatype and further attributes of the call are defined in the XML[36]. An advantage of decoupling the underlying I/O procedure is that the best-fitting implementation can be selected for a group of files.

Settings for the implementation, e.g., buffer size, can be defined without changing code. Moreover, data could be forwarded to a *visualization system*, simply discarded, or even multiple I/O methods can be selected to visualize and store data at the same time. Similar to SIONlib, the system is able to either write a shared file or to split logical I/O into several file system objects. With ADIOS, the *BP* file format is proposed that improves data locality for a single process and minimizes collisions between the processes.

[35] The communication and exchange phases are swapped for write operations.
[36] Elementary datatypes or arrays of arbitrary dimension are supported.

The API provides functions to the programmer to indicate when the computation starts or ends, or when the scientific application main loop occurs. On the one hand, this enables efficient communication to the servers without disturbing application communication. On the other hand, the pace in which data is created and written back is announced to the library. Concluding, ADIO provides a completely new API in which the programmer is forced to deal with I/O related aspects consciously – but due to the XML, system optimizations are possible without source-code modifications.

In our paper [KMKL11], the ADIOS interface is explained in detail. In this paper the interface is extended to offer visualization capabilities and improved energy efficiency.

Tuning library settings MPI libraries like IBM's *Parallel Environment* or Open MPI offer a rich set of environment specific parameters to tune the library internals towards a system or application. For example buffer sizes for the eager message protocol can be adjusted to the network characteristics. In Open MPI, the *Modular Component Architecture (MCA)* provides more than 250 parameters on a COST Beowulf cluster. The libraries provide empirically chosen defaults, which might be determined for a completely different system than the system the library is deployed on. Thus, the defaults might achieve only a fraction of theoretical performance. To provide a starting point for application specific tuning of those values, an administrator should provide appropriate values for the given system.

Chaarawi et.al. developed the *Open Tool for Parameter Optimization (OTPO)* for Open MPI which uses the automatic optimization algorithm from ADCL to determine the best settings of available MCA parameters for a given cluster system [CSGF08]. OTPO could be configured and run by administrators to set up efficient cluster defaults.

2.4. Performance Analysis and Tuning

In computer science, *performance analysis* refers to activity that fosters understanding in timing and resource utilization of applications. In terms of a single computer, the CPU, or to be more formal, each functional unit provided by the CPU, is considered to be a resource. Therefore, understanding resource utilization includes understanding run-time behavior and wall-clock time. For parallel applications, the concurrent computation, communication and parallel I/O increase the complexity of the analysis. Therefore, many components influence the resource utilization and run-time behavior; those have been discussed in Section 2.2.1.

Computational complexity theory is the field of computer science that provides methods to classify and estimate algorithm run-time depending on the problem size. Theoretical analysis of source code is usually infeasible as utilization of hardware at run-time can only be roughly estimated. Therefore, in practice, theoretical analysis is restricted to small code-pieces or clear application kernels, and typically software behavior is measured and assessed.

Programs can be classified according to their utilization characteristics and demand – important algorithms are categorized into 13 *motifs* [ABC+06]. Most applications could be thought of as a combination of the basic functionality required by those motifs. But even so, the characteristics of each real program must be analyzed individually.

In this section, it is first shown how application design and software engineering can assist in developing performance-demanding applications (Section 2.4.1). Those methods focus on integrated and automatic development to achieve efficient and performant applications.

As scientific programs usually require a huge amount of resources, one could expect them to be especially designed for performance. Unfortunately, that is not the case. One reason is that many scientific codes evolved over decades at a time when performance has been of low priority. Usually performance is analyzed after the correctness of the program has been evaluated. At this late stage, a functional version of the