Scalable Massively Parallel I/O to Task-Local Files

Wolfgang Frings
Jülich Supercomputing Centre
52425 Jülich, Germany
w.frings@fz-juelich.de

Felix Wolf
Jülich Supercomputing Centre
52425 Jülich, Germany
RWTH Aachen University
52056 Aachen, Germany
f.wolf@fz-juelich.de

Ventsislav Petkov
Technische Universität München
80333 Munich, Germany
petkovve@in.tum.de

ABSTRACT

Parallel applications often store data in multiple task-local files, for example, to remember checkpoints, to circumvent memory limitations, or to record performance data. When operating at very large processor configurations, such applications often experience scalability limitations when the simultaneous creation of thousands of files causes metadata-server contention or simply when large file counts complicate file management or operations on those files even destabilize the file system. SIONlib is a parallel I/O library that addresses this problem by transparently mapping a large number of task-local files onto a small number of physical files via internal metadata handling and block alignment to ensure high performance. While requiring only minimal source code changes, SIONlib significantly reduces file creation overhead and simplifies file handling without penalizing read and write performance. We evaluate SIONlib's efficiency with up to 288 K tasks and report significant performance improvements in two application scenarios.

1. INTRODUCTION

Driven by a rising demand for more computing power and accelerated by current trends in microprocessor design towards multicore chips, the number of processor cores on modern clusters and supercomputers is growing rapidly from generation to generation. While more than three quarters of the TOP500 systems employ at least two thousand cores, some machines at the top employ even more then a hundred thousand. With higher degrees of parallelism, efficient parallel file I/O becomes increasingly important, as file I/O can have a substantial impact on the overall application performance.

While offering optimizations for a variety of file access patterns, the particular strength of parallel file systems, such as GPFS [3, 8], Lustre [15, 16], and PVFS [20], is to provide efficient concurrent access to a single file via file striping across multiple disks and replicated I/O servers. However, due to historic file-system limitations, many applications still use one of the following two traditional approaches for parallel I/O, which may both adversely affect scalability [14].

The first method is called *single-file sequential* and uses one designated I/O task to access a single file on behalf of all others. While it works well on shared memory architectures, on machines with distributed private memory it typically requires gather and scatter operations to collect data from and distribute them to multiple tasks, respectively. In this scenario, file I/O is serialized and the bandwidth limited to what a single node can support. Since the designated I/O task has only limited memory capacity, multiple gather or scatter operations may be required while writing or reading the file incrementally, reducing the access performance even further. This method is often chosen when the different tasks own non-contiguous portions of the file, which can then be written in one large chunk.

In contrast, in the *multiple-file parallel* approach, which is often applied in message-passing programs, every task accesses its own file. This method is popular for storing task-local data such as restart (checkpoint) and scratch files or performance measurements, where the data belonging to individual tasks can be clearly separated. This is also why we refer to this type of I/O pattern as *parallel I/O to task-local files*. While offering performance advantages if the files reside on local disks, this method may not scale to tens of thousands of tasks in a shared file-system environment without local disks, which today's densely packed supercomputer architectures typically lack. Scalability problems of this approach may arise in the following two ways:

First, trying to create tens of thousands of files simultaneously in the same directory may be serialized due to contention at metadata servers. For example, on one of our test systems described later in this article, the parallel creation of 256 K files can take more than 30 minutes. Writing the files to separate directories is usually not a viable alternative, as it only shifts the problem to creating the directories. Albeit less expensive in terms of compute time, creating the files beforehand is inconvenient and requires maintaining some of the I/O functionality of an application separate from the main code. A script to generate the files during a preceding serial job would have to know their number, names, and locations, necessitating some form of agreement between the application and the script.

Second, even if such a separation can be tolerated, large numbers of files severely complicate file management tasks.

SC09 November 14-20, 2009, Portland, Oregon, USA.

For example, copying files to a tape archive (e.g., during backup) may be significantly slowed down. Especially when archival requests from different users are executed in an interleaved fashion, different files of the same directory may end up on different tapes, making their later retrieval challenging or even impractical if the tape cartridge has to be exchanged too often. Merging all of the files into a single file during a postprocessing step, for example using the tar command, also has its disadvantages both in terms of the time needed to perform the operation and the at least temporary duplication of the required storage space. Moreover, administering directories with tens of thousand of entries without support for group operations and automated filter tools appears ineffective. In addition to the increased complexity of managing large numbers of files, our experiences suggest that large-scale file operations can cause side effects including temporary service disruptions that are noticeable by arbitrary users and that can jeopardize the stability of the overall system. To avoid such phenomena, some environments impose limits on the total number of files a user or a group of users can have, offering another good reason not to use one physical file per task.

In this paper, we present an approach to allow applications with task-local parallel I/O, many of which naturally use multiple physical files, to take better advantage of parallel file systems. This is achieved by transparently mapping a large number of task-local logical files onto a single or a few physical files, solving both of the problems listed above. Our solution, which is implemented in an I/O library called SIONlib extending the ANSI C file I/O API, offers the following advantages:

- Simultaneous file creation becomes faster by orders of magnitude.
- Only minimal source-code changes are required, which mostly affect open and close operations. Metadata describing the extent and location of individual logical files are managed transparently.
- The read and write bandwidth remains unaffected. The alignment of logical files to file-system block boundaries avoids contention between any two logical files and ensures good bandwidth utilization.

The remainder of the paper is structured as follows. In Section 2, we review related work and explain why our solution is preferable for addressing the specific scenario introduced above. Then, we outline the SIONlib architecture in Section 3 with an emphasis on the organization of metadata, the programming interface, and operations on multifiles. In Section 4, we present a quantitative evaluation of SIONlib using up to 288 K tasks on two different machines and file systems. Subsequently, we demonstrate performance improvements in two real-world application scenarios in Section 5, including the checkpointing mechanisms of a simulation code and the tracing library of a performance tool. Finally, we draw our conclusion and discuss future perspectives in Section 6.

2. RELATED WORK

To make parallel I/O most efficient, knowledge of access patterns can be exploited to optimize the data flow between applications and disks, utilizing the parallelism available on hardware and software layers in between. The most prominent example of a platform-independent interface supporting parallel binary I/O is MPI I/O [17]. Using this library, data can be written collectively from all or a subset of the application tasks to a shared file, potentially taking advantage of hints including the number of disks to stripe files across, the stripe depth, or access patterns. Noteworthy is also MPI's support for shared I/O of non-contiguous distributed data. Every task can specify a non-contiguous view of a shared file, greatly simplifying the work with fined-grained data distribution schemes. Besides these more advanced features, MPI also offers all of the mechanisms needed to perform I/O in the traditional way, either following the single-file-sequential or multiple-file-parallel approach. While offering high-level functionality for strided and irregular access patterns, a transparent mapping of many logical task-local files onto few physical files is not directly supported, although it could be implemented using MPI I/O functions as lower-level routines. However, this would force the application to use MPI data types and an MPI-style programming interface, unnecessarily restricting the generality of our approach and potentially entailing more complex source-code changes in the application than needed. In this sense, we regard MPI I/O as orthogonal to our method.

Whereas MPI including its I/O substandard models data in terms of type maps, that is, as a list of basic data types placed at specific locations in an address space, high-level parallel I/O libraries, such as HDF5 [7] and NetCDF-4 [18], allow the reading and writing of data in terms of structured data models including annotated multidimensional arrays of typed elements and hierarchical groups of objects. The two libraries also store metadata describing the specific data format in addition to the actual data to facilitate easy sharing of files. Both libraries support the parallel reading and writing of their data sets, internally leveraging the MPI I/O layer. Whereas high-level parallel I/O libraries are useful for storing and retrieving structured scientific data, SIONlib is more suitable for binary stream data without any predefined structure. Similar to MPI I/O, using one of the high-level libraries instead of SIONlib would increase the transition cost by having to move to a more complex interface while offering no obvious performance advantages. Specifically, the need to define data structures before starting the actual I/O represents an extra burden for applications such as tracing tools that already use self-contained binary file formats.

Furthermore, ADIOS [13] provides an abstraction layer on top of various standard I/O interfaces ranging from low-level APIs such as simple POSIX I/O to MPI I/O and parallel higher-level APIs including the ones discussed above. Using this additional layer, an application can be easily configured to replace the underlying I/O transport method simply by modifying an XML configuration file. This improves flexibility when porting a code from one platform to another. Moreover, the data-group feature allows the selection of individual transport methods for different parts of the code to optimize the performance for a variety of file access patterns within the same application. In this context, SIONlib could serve as a further transport method to choose, further expanding the versatility of ADIOS. In fact, the ADIOS native binary file format employs concepts related to those underlying the design of the SIONlib format. First, it allows the

definition of process groups. A process group is the entire self-contained output from a single process that can be written independently into a contiguous disk space [10]. Second, a footer index ensures that the data section can grow beyond what is known at file creation time without moving data.

To optimize two-phase implementations of MPI collective I/O operations, Liao et al. [12] proposed dynamic file partitioning techniques that align the file domains assigned to aggregator processes with file-system lock boundaries. This helps avoid serialization through lock conflicts when multiple aggregator processes want to write in parallel and is similar in spirit to the block alignment used in SIONlib, only that SIONlib application processes write directly to the file without rearranging the file access pattern via aggregator processes.

Scalable operations on whole groups of files are defined by TBON-FS [2], a virtual file system that allows a client to efficiently communicate with a group of files via a tree-based multicast-reduction network. Extending familiar file-access idioms including file descriptors to groups, TBON-FS specializes in scalable operation request distribution and the aggregation of group file operation responses. Although making group operations more convenient by eliminating iteration across all group members, TBON-FS still operates on a potentially large number of physical files.

It remains an intriguing question why parallel systems themselves often do not provide better support for task-local I/O. According to our experiences, the main problem is not the aggregate bandwidth but the metadata-server contention that occurs when attempting to create large numbers of files in a single directory. Although the use of hashing to look up the file-system block designated for a certain directory entry brought some improvements [4, 22], the concurrent access to the file system blocks that contain the directory i-node more or less serializes this operation. SIONlib can handle this situation better only because it relies on superior knowledge of the intended access pattern, as we will see in the next section.

3. SIONlib

The objective of SIONlib is to make massively parallel I/O to task-local files such as checkpoints, scratch files, or log files more efficient. The basic concept of SIONlib is illustrated in Figure 1. Situated as an additional software layer between a parallel application and the underlying parallel file system, the main idea of SIONlib is to map a large collection of logical task-local files onto a single physical file (or at least a small number of files). This avoids contention at metadata servers during file creation without penalizing read and write bandwidth and simplifies file management operations such as listing a directory or copying the entire collection to a tape archive. In this sense, SIONlib can be thought of as a very simple application-level file system with an API and command-line utilities to access individual logical files. The programming interface of SIONlib is laid out as an extension of the ANSI C I/O interface, requiring only very few source code changes for applications already using ANSI C. Existing standard ANSI C read and write calls can be retained and the conversion of ANSI C file handles to numerical file descriptors for subsequent use in POSIX I/O calls remains possible. To allow parallel codes written in Fortran to take advantage of our library, a Fortran language mapping is supplied in addition to the C API. Although by design not tied to a specific parallel programming interface, the current version of SIONlib uses MPI for internal metadata exchange, which makes it particularly suitable for MPI codes. To meet its objectives, our approach exploits the following assumptions about the intended file access pattern:

- All task-local files can be created at the same time.
- Every file is accessed by only one task.

In addition, the maximum amount of data that may be written or read in one piece (i.e., in a single write or read call) by each individual task must be known in advance, at least if standard ANSI C read and write calls are to be used. In many cases, this limitation can already be addressed simply by choosing a generous maximum that can accommodate all foreseeable data sizes of a given application. To circumvent this restriction in a more systematic fashion, SIONlib offers its own version of read and write functions for binary data. Versions for formated text can be constructed in a similar way and will be provided in future versions of our library. Extrapolating from our experiences with the case studies presented in Section 5, we believe that the above assumptions are realistic for a broad range of applications, which could potentially benefit from using SIONlib.

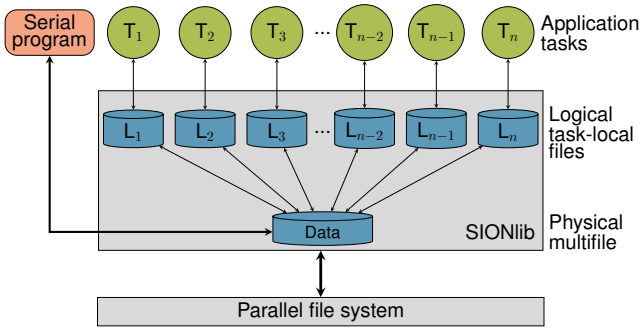
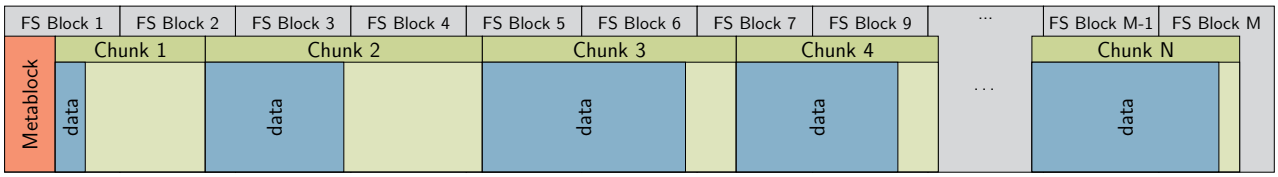


Figure 1: Basic concept of SIONlib: A large number of logical task-local files is mapped onto a single physical file (or a small set of physical files), which is called a multifile. The multifile can be accessed both from a parallel and a serial application.

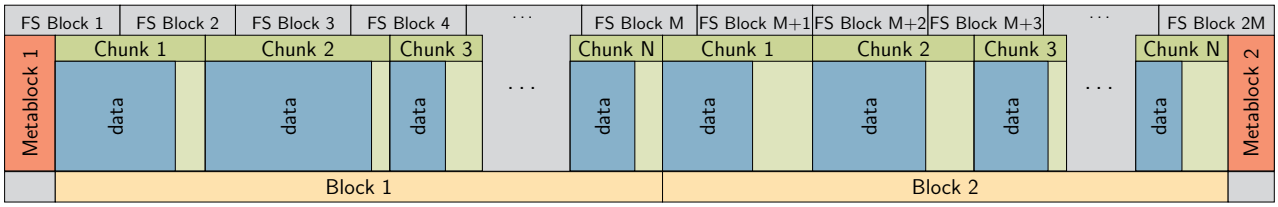
In the following, we explain the SIONlib file organization including the management of metadata, the programming interface to access task-local files in parallel, and a set of serial command-line utilities to perform operations including metadata dumping, defragmentation, and file splitting.

3.1 File Organization

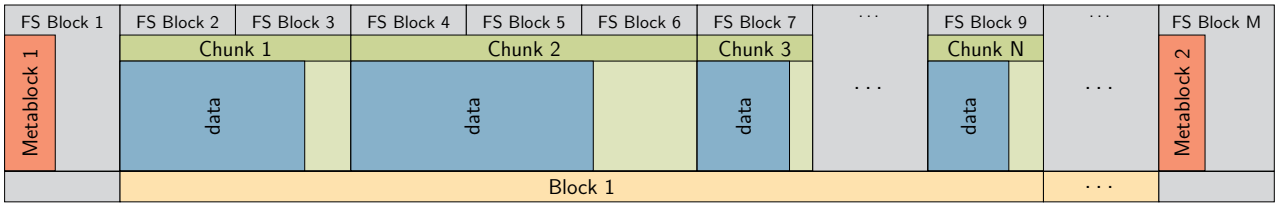
We motivate the SIONlib file organization step by step, starting from a very simple layout and refining it as we discuss new features. In the simplest case, the maximum (total) size of each individual task-local file is known in advance and they are all mapped onto a single physical file, which we call a *multifile*. The multifile is divided into so-called *chunks*, one for each task, as depicted in Figure 2(a). The size of each chunk corresponds to the maximum size requested by the task owning the chunk. The array of chunks is preceded



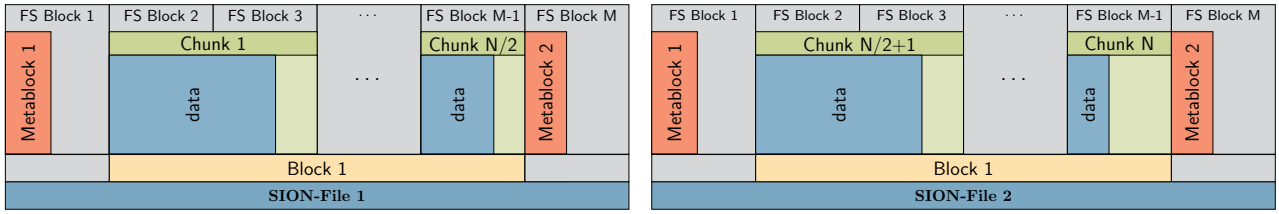
(a) The maximum size of task-local files is known in advance.



(b) Only the maximum amount of data written in one piece is known in advance.



(c) Chunks are aligned with file system block boundaries.



(d) Chunks are distributed across more than one physical file.

Figure 2: SIONlib file organization.

by a metadata block that specifies the start address of each chunk. Multifile creation is a collective operation, during which all tasks send their requested chunk size to a master task that is responsible for writing the metadata block and returning the individual start addresses to each task so that each of them knows where its reserved chunk begins. The file is closed via another collective operation, during which the master collects the number of bytes from each task that was effectively written and stores it in the metadata block. The close operation is again collective to avoid the inefficiency of having all tasks write to the metadata block concurrently.

However, the need to know the total amount of data written by each task may be too restrictive, as this knowledge is often not available when creating the files. Instead of knowing the total amount, we thus merely assume to know the maximum amount of data written in one piece by each task, leading to the layout depicted in Figure 2(b). The multifile is now organized in *blocks* with each block containing one chunk per task. If a task wants to write more bytes than left in the current chunk, it can request a new chunk of the same size. To ensure that every task knows the start address of every subsequent chunk allocated on its behalf without the need to communicate among tasks, an entire new block is allocated with a full array of chunks, again one for each task. Note that this may create substantial gaps in the multifile if only a subset of the tasks asks for additional chunks. However, since file systems tend not to physically allocate the empty blocks occurring in this scenario, the largest portion of these gaps exists only on the logical level. To avoid their later physical materialization, for example, when the multifile is copied in a certain way, the file can be defragmented in a postprocessing step (see Section 3.3). As we now need to store metadata indicating the space used in each chunk without knowing the total number of blocks (and chunks) in advance, we write the number of chunks per task and the space occupied by data in each of them to a second metadata block at the end of the multifile. Note that appending data to a multifile beyond the initially allocated space after it has been closed would require updating and moving the second metadata block. Although such a feature is currently not supported, adding it would not pose a fundamental design problem.

Having solved the problem of allocating sufficient file space to each task, we now need to make sure that every task can efficiently access its own portion of the file. Although there is no overlap between the chunks belonging to any two tasks,

adjacent chunks may nonetheless occupy parts of the same file-system block. With write locks being assigned at the granularity level of file-system blocks, this may cause lock contention when writing to those chunks. The situation is similar to the *false sharing* of cache lines in a multiprocessor. To avoid this performance-degrading side effect, the chunks are aligned with file-system block boundaries, and to avoid unnecessarily wasting space, the chunk size is chosen as a multiple of the file-system block size, as shown in Figure 2(c). Note that the block size of the target file system is determined automatically via the `fstat()` system call.

Our experiences suggest that in some environments using just a single physical file to store all logical task-local files may leave some of the hardware or software parallelism unused that is available between the application and the disks. For this reason, SIONlib also offers the option of distributing the logical files across a user-defined number of physical files (see Figure 2(d)). Every task is still mapped onto a single physical file, but two tasks may now end up being mapped onto different physical files. In the remainder of the paper, the term multifile refers to the entire collection of underlying physical files if multiple physical files are used. In addition to the number of physical files, the user can also influence the exact mapping of application tasks to physical files, for example, to allocate one physical file per I/O node on Blue Gene if desired.

3.2 Application Programming Interface

The SIONlib API is designed as an extension of the ANSI C file I/O API, demanding only very little source-code changes for applications that already use ANSI C I/O to write multiple task-local files in parallel. In the simplest case, changing the application to write a SIONlib multifile only requires replacing the open and close calls, as we will see below. Multifiles with multiple underlying physical files are handled transparently. SIONlib supports the following four modes of accessing a multifile:

- Parallel write
- Parallel read
- Serial write
- Serial read

3.2.1 Parallel write

This mode is the default mode when writing logical task-local files from a parallel application. Both open and close calls are collective operations (Listing 1). The open call takes the chunk size (i.e., the maximum number of bytes expected to be written in one piece) as a parameter, which can be individually chosen for each task. The global communicator `gcom` includes all of the tasks for which a logical file needs to be created. The local communicator `lcom` defines a subset of the tasks that share the same underlying physical file. The operation returns two file handles: (i) a normal ANSI C file handle to the task-local file to be used in subsequent ANSI C write operations just as though the logical file were a physical file and (ii) a SIONlib file handle to be used in subsequent calls to the SIONlib API. The call to `sion_ensure_free_space()` is only needed if the number of bytes to be written may exceed the available space in the current chunk so that a new chunk must be allocated. In this case, the file pointer is advanced to the start of the new chunk. Writing the data itself then occurs via a call to `fwrite()`, as if writing to a physical task local file. If a need arises to write more bytes than a single chunk can accommodate, the combination of ensuring free space and writing the data should be replaced with a single call to `sion_fwrite()`. This splits the data internally into smaller pieces so that chunk boundaries are observed. In this way, the above-mentioned restriction of having to know the maximum amount of data written in one piece can be relaxed. If the use of POSIX `write()` is preferred to ANSI C `fwrite()`, the ANSI C file handle can be converted into a numerical file descriptor as usual. In addition, all three options can be mixed freely. This applies to the other multifile access modes accordingly.

```
/* open collective */
sid=sion_paropen_mpi( ... ,&chunksize,
                          gcom,
                          &lcom,
                          &fileptr, ...);

/* write non-collective */
sion_ensure_free_space(sid, nbytes);
fwrite(data, 1, nbytes, fileptr);
/* or */
sion_fwrite(data, 1, nbytes, sid);

/* close collective */
sion_parclose_mpi(sid);
```

Listing 1: Parallel write.

3.2.2 Parallel read

Reading the multifile in parallel is similar to writing it (Listing 2). Again, open and close are collective operations, whereas the actual reading can occur in isolation. A call to `sion_feof()` ensures that the end of the file has not yet been reached. Like in the previous case, the user has two choices: either (i) reading within the limits of the current chunk using `fread()`, with the limit being enforced by a preceding call to a SIONlib guard function to identify the number of bytes left in the chunk, or (ii) reading without limit using the customized read function `sion_fread()`.

```
/* open collective */
sid=sion_paropen_mpi( ... ,&chunksize,
                          gcom,
                          &lcom,
                          &fileptr, ...);

/* read non-collective */
if (!sion_feof(sid)) {
  btoread=sion_bytes_avail_in_chunk(sid);
  bread=fread(localbuffer, 1, btoread, fileptr);
  /* or */
  sion_fread(localbuffer, 1, nbytes, sid);
}

/* close collective */
sion_parclose_mpi(sid);
```

Listing 2: Parallel read.

3.2.3 Serial write

In addition to writing a multifile from a parallel application, the programming interface also offers functions to write a multifile from a serial application (Listing 3), a necessary

prerequisite to build serial postprocessing tools. Since the open call is now executed by only one process, a whole array of chunk sizes needs to be supplied as a parameter. The `sion_seek()` call helps to navigate within the multifile, allowing the user to conveniently locate a specific position within a given chunk of a given task (i.e., rank).

```
sid=sion_open( ..., &chunksizes, &fileptr);

sion_seek(sid, rank, chunk, pos);
sion_ensure_free_space(sid, nbytes);
fwrite(..., fileptr);

sion_close(id);
```

Listing 3: Serial write.

3.2.4 Serial read

Serial reading can happen either with a task-local or a global view. The local view is convenient for extracting the portion belonging to a single task only, whereas the global view is needed to read the data of all tasks, for example, when calculating global statistics. To open a multifile in the local-view mode, the rank of the task is supplied as an argument to the open operation (Listing 4). The actual reading is done in the same way as in the parallel case.

```
sid=sion_open_rank( ..., rank, &fileptr);

/* reading like in the parallel case */

sion_close(sid);
```

Listing 4: Serial read with task-local view.

If a multifile is opened in the global-view mode (Listing 5), the user usually first needs to retrieve all of the metadata to learn about the number of tasks (i.e., ranks), the number of chunks per task, and the chunk sizes used by individual tasks, etc.. Using the metadata information, a meaningful seek target can be chosen as starting point for a subsequent read operation.

```
sid=sion_open( ...,&fileptr);
sion_get_locations(sid, ...,
                   &nrranks,
                   &nrchunks,
                   &chunksizes,
                   ...);

sion_seek(sid, rank, chunk, pos);
fread(..., fileptr);

sion_close(sid);
```

Listing 5: Serial read with global view.

3.2.5 Fortran interface

Taking into account the fact that numerous scientific codes are written in Fortran, we also provide a Fortran language mapping in addition to the C API to make SIONlib more widely applicable. The Fortran interface essentially mirrors the C interface with the exception that read and write operations must use the SIONlib functions, potentially requiring slightly more source code changes.

3.3 Command-Line Utilities

The current version of SIONlib provides three command-line utilities to analyze, split, and defragment multifiles.

- The dump tool prints the multifile metadata to the standard output. This is a convenient way to learn more about the structure of the multifile to see, for example, how many logical files it contains and how large they are.
- The split tool extracts all or only distinct logical files from a given multifile and recreates the corresponding physical files.
- The defragment tool generates a new multifile from an existing one, contracting all of the blocks into a single block, that is, the new file contains only one chunk per task with the data from all chunks of this task found in the input file. In addition, all gaps in the form of unused file-system blocks are removed.

4. QUANTITATIVE EVALUATION

In this section, we evaluate the efficiency of our approach by measuring the time needed for basic file operations, also comparing parallel I/O to physical task-local files against the logical file mapping provided by SIONlib. After underlining our claim that parallel I/O to physical task-local files does not scale to large numbers of tasks, we examine SIONlib's performance under the influence of different parameters. All of our measurements were performed on the two systems described below.

Jugene. An IBM Blue Gene/P system located at the Jülich Supercomputing Centre in Germany [11]. Each of the 72 racks has 1024 compute nodes containing a 4-way SMP 32-bit PowerPC 450 with a clock rate of 850 MHz. The total number of cores is 294,912 and the overall peak performance is 1003 teraflops. The 608 I/O nodes are connected via 10GigEthernet to a file server running GPFS Version 3.2.1 and consisting of 12 IBM p570 Power6 8-way SMP nodes. The server offers access to a SAN-attached disk capacity of 1.1 PB. The maximum bandwidth to the scratch file system where we conducted our experiments is 4.7 GB/s. GPFS allows all nodes to perform file metadata operations, not relying on a centralized metadata server. Metadata are managed at the node using the file or in the case of parallel access to the file, at a dynamically selected node among the ones using the file.

Jaguar. A Cray XT4/5 system located at the Oak Ridge National Laboratory in the US [19]. The XT4 partition used for our experiments has a total number of 7,832 quad-core 2.1 GHz AMD Opteron nodes. The total number of cores is 31,328 and the aggregate system performance is approximately 263 teraflops. Jaguar is attached to a Lustre file system Version 1.6.5 with a scratch file-system capacity of 600 TB split into three file systems. The file server configuration of the scratch file system partition used in this study includes 72 object storage targets (OSTs) and 3 metadata servers (MDSs) with dual-core 2.6 GHz AMD Opteron processors, which are connected via Fibre-Channel. The overall file-system bandwidth is 40 GB/s. In contrast to GPFS,

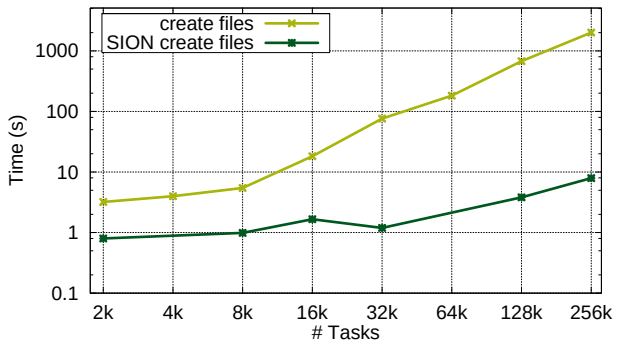
Lustre uses dedicated metadata servers. Moreover, Lustre allows the stripe factor (i.e., the number of OSTs a file is distributed across) and the stripe depth (i.e, block size) to be configured on a per-file or per-directory basis.

Note that the relatively expensive measurements presented in this section were taken on the two systems under normal production conditions. Although the reported numbers represent averages of typically three to five measurements to compensate for natural run-to-run variations, we still expect them to represent snapshots and general performance trends rather than precisely reproducible numbers due to the enormous variations common for I/O operations. However, we believe that this is still sufficient to support our hypothesis.

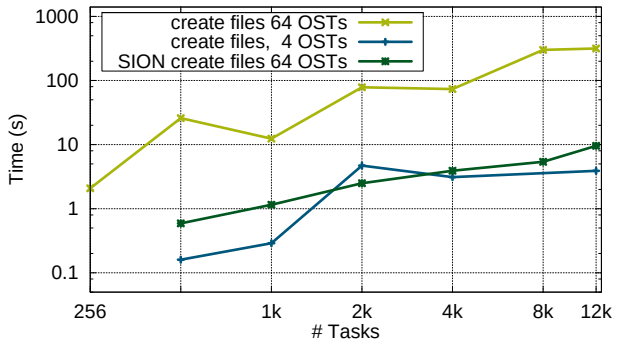
4.1 Creating Multiple Files in Parallel

The following experiments compare the parallel creation of large numbers of physical files in the same directory with the creation of a SIONlib multifile consisting of a smaller number of physical files. Whereas in the first case the time between the first open and the last close is measured, creating a SIONlib multifile also involves several MPI collective operations to broadcast the file-system block size, to gather chuck sizes, and to scatter file offsets in addition to writing the first metadata block (i.e., the header).

As shown by the graphs in Figure 3, the time needed to create multiple physical task-local files in the same directory in parallel increases with the number of tasks (i.e., files).



(a) Jugene.



(b) Jaguar.

Figure 3: Performance of creating new task-local files in parallel in the same directory.

Although the way our library is implemented caused the multifile creation time to grow with the number of tasks as well, for the investigated configurations the time needed to create a multifile was at least by an order of magnitude below the corresponding time needed to create a separate physical file per task.

Looking at the highest measured configuration on Jugene, the parallel creation of 256 K files took approximately 33 min, which clearly demonstrates the scalability limits of using multiple task-local files in parallel. In contrast, having 256 K tasks create a multifile (16 physical files) took only 8 s. On Jaguar, the time needed for the parallel creation strongly depends on how many OSTs have been chosen to stripe the files across. Since opening a file affects all of the OSTs involved, the time increases with the number of OSTs. Whereas creating 12 K files in parallel took only 4 s with the default configuration of 4 OSTs, which does not seem critical at this scale, it took 5 min with 64 OSTs. We therefore performed a comparison for the 64-OST case only, where creating a multifile (32 physical files) on 12 K cores took about 10 s. This constitutes a more than 30-fold improvement. Unfortunately, time constraints did not allow us to perform larger tests on this system.

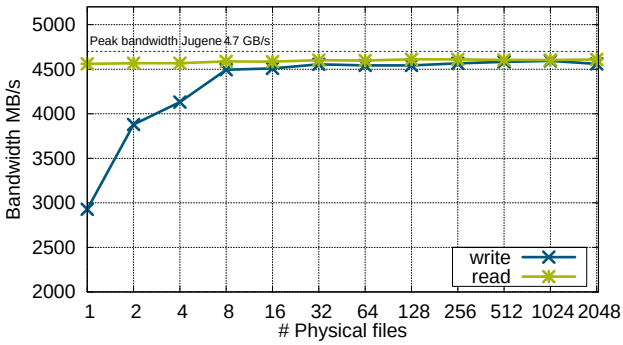
4.2 Bandwidth

While our approach significantly reduces the file creation overhead, as demonstrated in Section 4.1, it is also important that SIONlib's logical file mapping does not incur any bandwidth penalty. If the available bandwidth cannot be reasonably utilized, increasing tasks numbers and problem sizes will be confronted by the problem of I/O consuming growing fractions of the overall runtime. For this reason, we compare the bandwidth achieved with SIONlib to the maximum bandwidth available on the system and to the bandwidth achieved when writing to or reading from physical task-local files. However, before drawing those comparisons, we examine the influence of the number of underlying physical files and of the file-system block alignment on the bandwidth achievable with SIONlib.

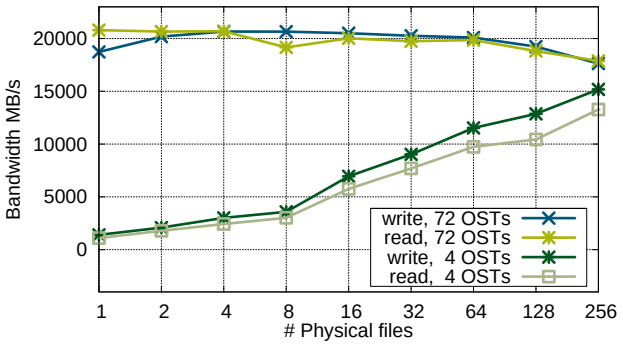
4.2.1 Multiple Physical Files

Since using a single underlying physical file to store the contents of a SIONlib multifile may not offer the best bandwidth utilization possible on a given system and systems may also differ with respect to the optimal number of underlying physical files, SIONlib was designed in such a way that this number can be freely configured. Figures 4(a) and 4(b) show bandwidth measurements for different numbers of files. Apparently, at least GPFS rewards the distribution of the data across multiple physical files.

On Jugene, the measurements were taken on the full system, increasing the number of physical files from 1 to 2048. Since the bandwidth of the file system is limited to about 4.7 GB/s, a saturation of the performance gain could be observed between 8 and 16 physical files. A potential reasons for the lack of bandwidth if using less than 8 files may be found in the way GPFS manages the metadata blocks that contain information on the actual data blocks of a file. Since only a single GPFS client is responsible for accessing the metadata blocks of a given file, this client may turn into



(a) Jugene (64 K tasks, 2 TB data)



(b) Jaguar (2 K tasks, 1 TB data)

Figure 4: Bandwidth when using multiple underlying physical files with SIONlib.

a bottleneck if the number of tasks writing to the file grows very large.

To further investigate the relationship between bandwidth and the number of underlying physical files, we exploited the fact that on Jaguar striping parameters can be adjusted and ran our test with two different sets of striping parameters. The first configuration is the default setting, which stripes a file across 4 OSTs and uses a stripe depth of 1 MB. The second one is better suited for parallel I/O to a single file, striping a file across 72 OSTs with a stripe depth of 8 MB. Whereas the default setting shows a steady bandwidth increase as the number of physical files is raised to about 256 files, the optimized configuration delivered good performance already for a single file and no significant benefits were observed when using more than one file. In fact, we noticed a slight bandwidth drop off when moving from 128 to 256 files.

Regardless of the number of files, using a higher number of OSTs was always superior to the unoptimized configuration. This suggests that on Jaguar, choosing the right striping pattern is as important as choosing the number of files. Nevertheless, using more than one physical file is necessary if the size of a single file is limited. Typically, the data sets written by large-scale parallel applications can be in the range of multiple TB. For further measurements, we decided to use at least 16 physical files on both systems.

4.2.2 *Block Alignment*

SIONlib aligns task-local chunks with file-system block boundaries to avoid contention, which may occur if two or more tasks simultaneously write to the same file-system block because both chunks occupy a portion of it. To show the benefit of the alignment, we ran two tests on Jugene, writing and reading data to/from the GPFS file system, which is configured with a block size of 2 MB. In the first instance, we configured SIONlib with the correct block size so that the data was perfectly aligned. In the second instance, we configured SIONlib with a block size of 16 KB so that chunks of different tasks would share the same file-system block. Table 1 shows results with 32K tasks and 16 underlying physical files on Jugene. The numbers clearly show that contention diminishes the write bandwidth roughly by a factor two so that block alignment is strongly recommended on Jugene. One reason for this considerable difference is that the smallest granularity level at which files can be locked for write access in GPFS is the file-system block. Assuming that reading the files would use sharable locks, we have currently no explanation for the improvement of the read bandwidth though. In contrast, preliminary tests on Jaguar have not yet confirmed a significant influence of block alignment on the bandwidth.

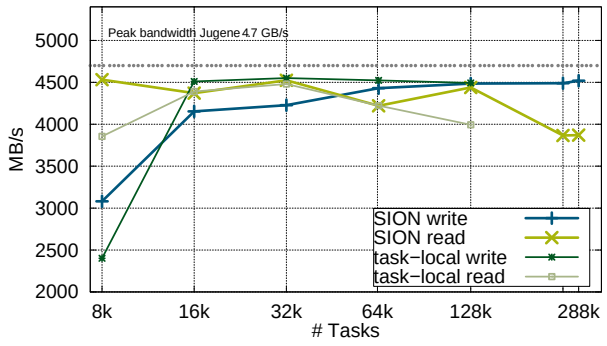
#tasks	data size	blksize	write	read
32768	256 GB	2 MB	3650.2 MB/s	3544.5 MB/s
32768	256 GB	16 KB	1863.8 MB/s	2000.9 MB/s
			→ 1.95x	→ 1.771x

Table 1: Bandwidth to a SIONlib multifile with 16 underlying physical files on Jugene with (top) and without (bottom) block alignment.

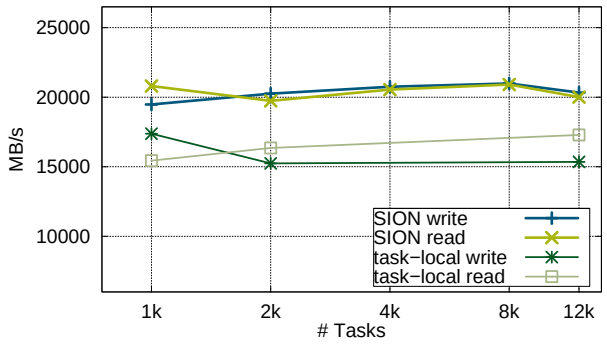
4.2.3 *Comparison to Physical Task-Local Files*

Figure 5 compares the bandwidth of SIONlib using 32 underlying physical files with the bandwidth of traditional parallel I/O to physical task-local files on a range of scales. On Jugene, we conducted tests with SIONlib on up to 288 K cores, while time limitations prevented us from conducting tests without SIONlib on more than 128 K cores. On both systems, SIONlib was configured to match the automatically detected block size of 2 MB of the scratch file system and the overall size of the SIONlib multifile was 2 TB. To avoid cache effects, the process-to-data mapping was changed between writing and reading – similar to the approach applied in IOR [9]. On Jaguar, we used 72 OSTs for SIONlib, as this value was found to work efficiently during our measurements presented in Section 4.2.1. In contrast, we used only 4 OSTs for traditional parallel task-local I/O since (i) according to our experiences using a higher number reduces the bandwidth presumably due to an excessive number of files an OST must handle in parallel and (ii) a long file creation time suggests to use a low number in this case anyway.

With and without SIONlib, the bandwidth was saturated with 16 K or more tasks on Jugene. On Jaguar, write and read bandwidth was always significantly better when using SIONlib, but in neither of the two cases we were able to reach the generous peak bandwidth of 40 GB/s.



(a) Jugene



(b) Jaguar

Figure 5: Bandwidth of SIONlib I/O with 32 underlying physical files in comparison to parallel I/O to physical task-local files.

5. CASE STUDIES

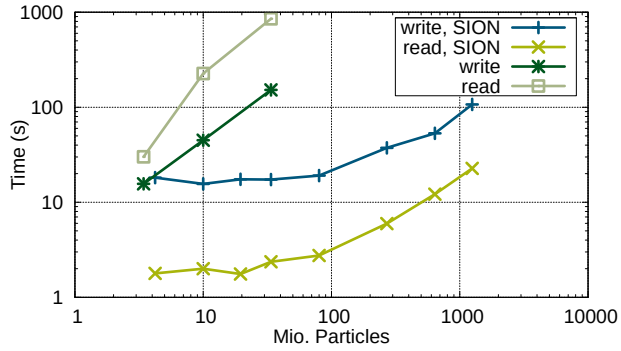
To present evidence of SIONlib's usefulness in practice, we integrated SIONlib into two real-world applications. The first one is the mesoscopic particle dynamics simulation MP2C [24], the second one is the performance analysis tool Scalasca [6, 21]. In both cases, we can report substantial performance improvements.

5.1 MP2C

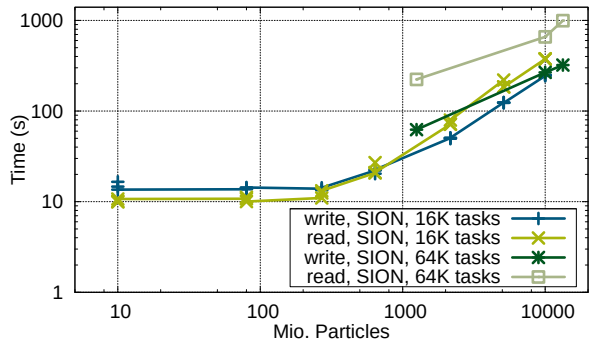
Mesoscale simulations of hydrodynamic media bridge the gap between microscopic simulations on the atomistic level and macroscopic simulations on the continuum level. To study colloidal suspensions or semi-diluted polymer systems, the Fortran code MP2C couples multiple-particle collision dynamics, an established mesoscale simulation approach, with molecular dynamics. The current version of MP2C is based on MPI and uses a domain decomposition approach, where geometrical domains of the same volume are distributed across the different processes.

Due to the extremely large numbers of particles involved, the simulation of realistic system sizes on long time scales requires an efficient implementation of the simulation code. Although the basic algorithm used in MP2C was shown to scale well, a limiting factor in production runs was met in file I/O operations used to write checkpoint/restart and particle trajectory files. To avoid file handling issues that arise from having a potentially large number of files from the very beginning, the authors of the code had originally decided to follow the single-file sequential approach explained in Section 1, where one designated I/O task writes a single file on behalf of all others. Experiencing all the scalability limitations of this approach ranging from serialized I/O in combination with alternating gather and write operations, the problem size that could be used for MP2C on 1 K cores of Jugene was effectively limited to roughly 10 M particles. Since having each task write its restart data to a separate physical file was no option due to the issues discussed earlier, we found MP2C to be a suitable candidate for SIONlib. After modifying approximately 50 lines of code, which involved making each task write its restart data to its own logical file instead of funneling it to the central writer task, the application could run problem sizes of more than one billion particles.

Figure 6(a) compares the times needed by MP2C to write and read restart files on 1 K cores of Jugene with and without using SIONlib, respectively. The measurements were taken on a single rack in SMP mode. The 1000 task-local files were mapped onto a single physical file. Since SIONlib writes at least one file-system block per task to accommodate the 56 bytes per particle, the advantage of using our approach materializes only for larger problem sizes, where they are significant. For 33 M particles, the I/O performance was improved by 1-2 orders of magnitude.



(a) 1 K cores (1 physical file)



(b) 16 K and 64 K cores (16 physical files)

Figure 6: Times needed by MP2C for writing and reading restart files on 1 K cores of Jugene with and without using SIONlib and on 16 K and 64 K cores with SIONlib only.

Beyond performance improvements for 1 K cores, adopting our library allowed MP2C to scale to much larger configurations than before – both in terms of the number of particles and the number of cores. Figure 6(b) shows the times needed for restart file I/O on 16 K and 64 K cores, respectively. All of the task-local files were mapped onto 16 physical files. Note that the restart data written for 10 billion particles consume 0.5 TB of disk space, explaining the still relatively large amount of I/O time. Thanks to SIONlib, production runs now simulate up to 0.5 billion particles.

5.2 Scalasca

Scalasca is an open-source toolset that can be used to analyze the performance behavior of parallel applications and to identify opportunities for optimization. It has been specifically designed for use on large-scale systems including IBM Blue Gene and Cray XT, but is also well-suited for small- and medium-scale HPC platforms. As a distinctive feature, Scalasca provides the ability to identify wait states in a program that occur, for example, as a result of unevenly distributed workloads, by searching event traces for characteristic patterns. Especially when trying to scale communication-intensive applications to large processor counts, such wait states can present severe challenges to achieving good performance. The trace analysis is available for MPI applications and is currently being extended towards support for the hybrid MPI/OpenMP programming model. To perform a pattern search, each task first records local events in a collection buffer and writes them to a task-local file at measurement finalization according to the multiple-file parallel method. Following the workflow depicted in Figure 7, the traces are then loaded postmortem into the distributed memory of a parallel trace analyzer program. Although the completion of trace analyses for applications running on up to 64 K cores has already been demonstrated [6], the experiment activation (i.e., creating the trace files and initializing the tracing library) was found to be a notable bottleneck, as one would expect.

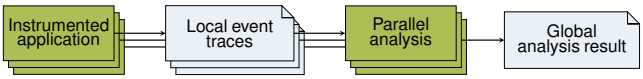


Figure 7: Parallel trace analysis in Scalasca.

Table 2 shows measurement activation times on Jugene before and after the integration of SIONlib, which required changing less than 50 lines of C code in the Scalasca tracing module. The times were obtained from running the fully instrumented MPI version of the ASC SMG2000 benchmark [1] on 64 K cores, using a 10x10x10 problem size per process. This time, we specified 16 underlying physical files to accommodate the aggregate trace size of 1105 GB. As can be seen, the activation time was reduced by a factor of 9.8 to 28.1 s with the pure file creation consuming roughly 1 s. The write bandwidth was even slightly improved.

To transparently retain the zlib [5] compression used by the Scalasca tracing module during the write operation, a chunk size equal to the amount of uncompressed data was chosen so that only one block of chunks needed to be written. In addition to the changes to Scalasca itself, reading the traces into the trace analyzer, which makes parallel use of the serial interface in the task-local view mode, required a minor customization of the zlib read function to ensure that the end of the local chunk was recognized. Moreover, to ensure good write performance on Jugene, the zlib write function had to be slightly adapted so that the incremental write it performed once the compression buffer had been filled observed the boundaries of GPFS file-system blocks. The modifications of the zlib library did not affect more than 10 lines of code in total. In the medium term, we plan to support the analysis of hybrid codes via a separate multifile for every OpenMP thread identifier, resulting in at most four multifiles on Jugene with its four cores per node, working around the currently still somewhat MPI-centric interface of SIONlib.

I/O type	#tasks	trace size	activation	write BW
Task-local	65536	1105 GB	253.5 s	3476 MB/s
SIONlib	65536	1105 GB	25.9 s	3589 MB/s
			→ 9.8x	

Table 2: Scalasca trace measurement activation time without (top) and with (bottom) SIONlib for a 64 K core run of SMG2000.

6. CONCLUSION

This work addresses a common scalability problem of parallel I/O to task-local files on peta-scale systems that is manifested in (i) a prolonged file creation overhead and (ii) the difficulty of managing excessive numbers of files. The I/O library SIONlib described in this paper solves the two problems by transparently mapping a large number of logical task-local files onto a very small number of physical files via internal metadata handling. In this way, the time needed for the parallel creation of tens of thousands of task-local files can be reduced from several minutes to just a few seconds. As we have demonstrated in two application scenarios, a key advantage of SIONlib is that adapting an application to use our library requires very little source-code changes. In addition to its ease of use, the alignment of task-local chunks with file system block boundaries makes sure that no penalty has to be paid in terms of read or write bandwidth. To allow a broad range of applications to take advantage of SIONlib, a fully documented version has been made available to the community for download under an open-source license at [23].

Our benchmark evaluation underlined our hypothesis of no bandwidth penalty on both target architectures, even showing some improvements on Jaguar. With respect to file creation time, SIONlib demonstrated clear advantages on Jugene with up to 256 K tasks, while the small configurations we tested on Jaguar (up to 12 K tasks) established substantial improvements only for larger numbers of OSTs. Tests on larger Jaguar partitions will be needed to further illuminate this relationship. It is safe to say, however, that using SIONlib usually increased and almost never reduced performance. Most important, it always avoided the creation of an excessive number of physical files and the management problems that such excessive numbers may entail.

While not knowing the maximum amount of data read or written in one piece only slightly reduces the convenience of using our library, another limitation of our file layout is

that the maximum number of tasks must always be known in advance, posing challenges for dynamic process management. Moreover, the current interface has been primarily designed for MPI applications, so that thread-local data in hybrid codes have to be managed at the application level. More systematic support for multithreaded applications is therefore already on our road map. Furthermore, failures, such as premature application termination or file quota violation, may cause the second metadata block to be lost. To improve SIONlib's robustness in such an event, we plan to add small pieces of metadata to each chunk so that the full metadata can be restored if needed. Finally, we are contemplating the addition of transparent file compression to SIONlib (e.g., via integrating zlib) to avoid customizations such as the one described in the context of Scalasca.

Putting this work in a broader perspective, we hope that our observations regarding the scalability of task-local file I/O will raise the awareness for this problem in the wider HPC community and that the general ideas developed in this paper for its solution, which have been validated using SIONlib as a reference implementation, will ultimately be adopted by designers of standard file systems and I/O libraries.

Acknowledgments

This work was partially funded under the German Helmholtz Association Young Investigators Program under Contract No. VH-NG-118. This research also used resources of the Jülich Supercomputing Center and resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725. Finally, we would like to thank Godehard Sutmann and Jens Freche for giving us access to the MP2C code and for offering their advice during the integration of SIONlib.

7. REFERENCES

[1] Advanced Simulation and Computing Program. The ASC SMG2000 benchmark code, 2001. `https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/smg/`.

[2] M. J. Brim and B. P. Miller. Group file operations for scalable tools and middleware. Technical Report No. TR1638, Computer Sciences Department, University of Wisconsin, 2008.

[3] S. Fadden. An introduction to GPFS version 3.2.1, November 2008. IBM Corporation.

[4] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing – a fast access method for dynamic files. *ACM Transactions on Database Systems (TODS)*, 4(3):315 – 344, 1979.

[5] J. Gailly and M. Adler. zlib general-purpose compression library, version 1.2.3. `http://www.zlib.net`, 2005.

[6] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. A scalable tool architecture for diagnosing wait states in massively-parallel applications. *Parallel Computing*, 35(7):375–388, 2009.

[7] HDF5. `http://www.hdfgroup.org/HDF5/`.

[8] IBM. General Parallel File System. `http://www-03.ibm.com/systems/clusters/software/gpfs/index.html`.

[9] IOR Parallel File System Benchmark. `http://sourceforge.net/projects/ior-sio/`.

[10] C. Jin, S. Klasky, S. Hodson, J. Lofstead, F. Zheng, M. Wolf, and R. Ross. *ADIOS User's Manual*. Oak Ridge National Laboratory, November 2008.

[11] Jülich Supercomputing Centre. JUGENE. `http://www.fz-juelich.de/jsc/jugene`.

[12] W. Liao and A. Choudhary. Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols. In *Proc. of the ACM/IEEE SC08 Conference*, Austin, TX, November 2008.

[13] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proc. of the 6th International Workshop on Challenges of Large Applications in Distributed Environments (CLADE)*, pages 15–24, Boston, MA, USA, 2008.

[14] J. M. May. *Parallel I/O for High Performance Computing*. Morgan Kaufmann Publishers, 2001.

[15] S. Microsystems. Lustre file system. `http:www.lustre.org`.

[16] S. Microsystems. Lustre file system – high performance storage architecture and scalable cluster file system (white paper), October 2008. `http://www.sun.com/software/products/lustre/docs/lustrefilesystem_wp.pdf`.

[17] MPI Forum. MPI: A message passing interface standard, version 2.1. Chapter 13, September 2008.

[18] NetCFD. `http://www.unidata.ucar.edu/software/netcdf/`.

[19] Oak Ridge National Laboratory. Jaguar. `http://www.nccs.gov/computing-resources/jaguar/`.

[20] Parallel Virtual File System. `http://www.pvfs.org/`.

[21] Scalasca. `http://www.scalasca.org/`.

[22] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, pages 231–244, Berkeley, CA, USA, 2002. USENIX Association.

[23] SIONlib. `http://www.fz-juelich.de/jsc/sionlib/`.

[24] G. Sutmann, R. G. Winkler, and G. Gompper. Multi-particle collision dynamics coupled to molecular dynamics on massively parallel computers. (in preparation).