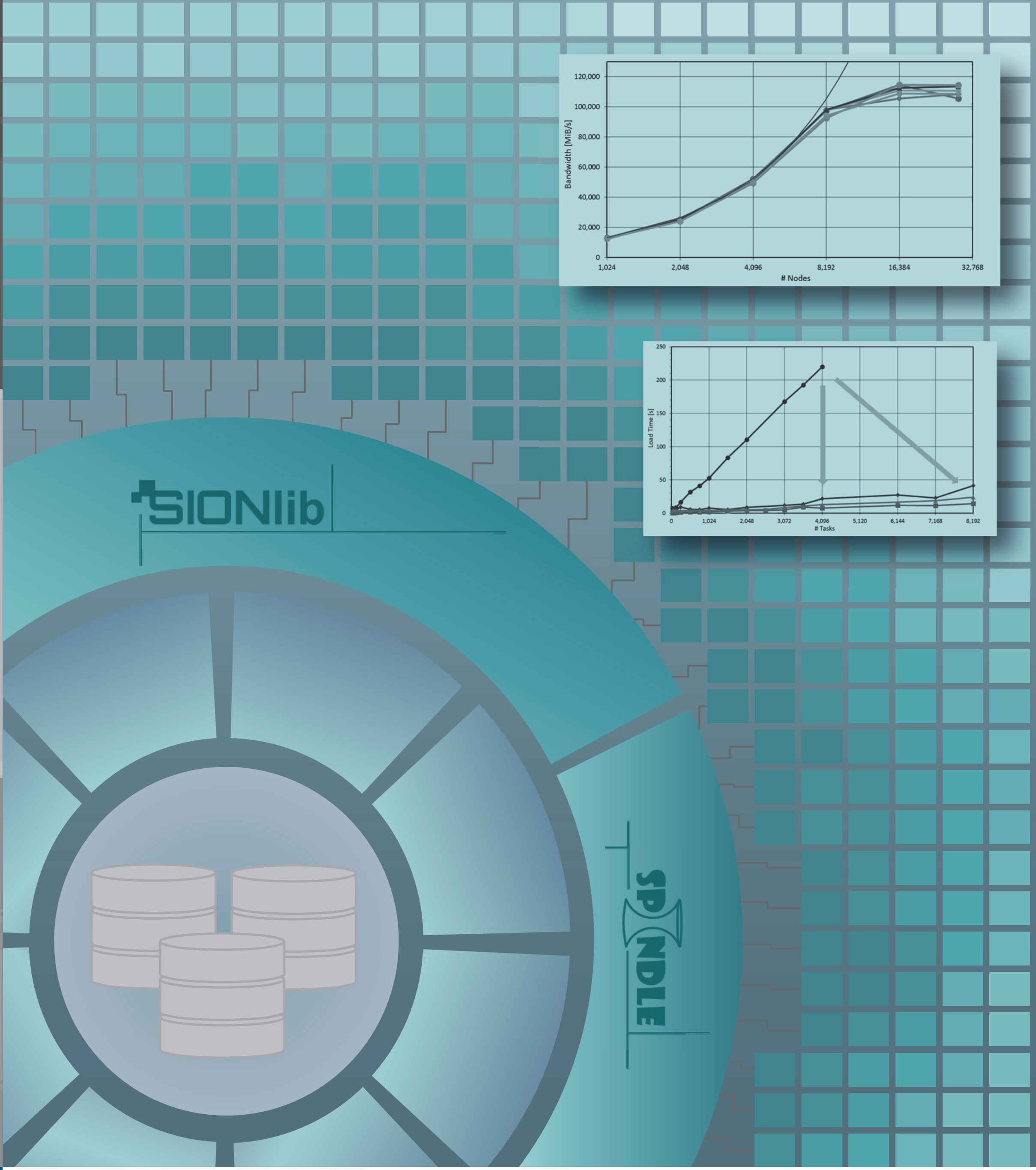


Efficient Task-Local I/O Operations of Massively Parallel Applications

Wolfgang Frings

JÜLICH
FORSCHUNGSZENTRUM

Schriften des Forschungszentrums Jülich
IAS Series Band 30

Forschungszentrum Jülich GmbH
Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)

Efficient Task-Local I/O Operations of Massively Parallel Applications

Wolfgang Frings

Schriften des Forschungszentrums Jülich
IAS Series Band 30

ISSN 1868-8489 ISBN 978-3-95806-152-1

Bibliografische Information der Deutschen Nationalbibliothek.
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der
Deutschen Nationalbibliografie; detaillierte Bibliografische Daten
sind im Internet über http://dnb.d-nb.de abrufbar.

Herausgeber und Vertrieb:	Forschungszentrum Jülich GmbH Zentralbibliothek, Verlag 52425 Jülich Tel.: +49 2461 61-5368 Fax: +49 2461 61-6103 E-Mail: zb-publikation@fz-juelich.de www.fz-juelich.de/zb
Umschlaggestaltung:	Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH
Druck:	Grafische Medien, Forschungszentrum Jülich GmbH
Copyright:	Forschungszentrum Jülich 2016

Schriften des Forschungszentrums Jülich
IAS Series Volume 30

D 82 (Diss. RWTH Aachen University, 2016)

ISSN 1868-8489
ISBN 978-3-95806-152-1

Persistent Identifier: urn:nbn:de:0001-2016062000

Abstract

Applications on current large-scale HPC systems use enormous numbers of processing elements for their computation and have access to large amounts of main memory for their data. Nevertheless, they still need file-system access to maintain program and application data persistently. Characteristic I/O patterns that produce a high load on the file system often occur during access to checkpoint and restart files, which have to be frequently stored to allow the application to be restarted after program termination or system failure. On large-scale HPC systems with distributed memory, each application task will often perform such I/O individually by creating task-local file objects on the file system. At large scale, these I/O patterns impose substantial stress on the metadata management components of the I/O subsystem. For example, the simultaneous creation of thousands of task-local files in the same directory can cause delays of several minutes. Also at the startup of dynamically linked applications, such metadata contention occurs while searching for library files and induces a comparably high metadata load on the file system. Even mid-scale applications cause in such load scenarios startup delays of ten minutes or more. Therefore, dynamic linking and loading is nowadays not applied on large HPC systems, although dynamic linking has many advantages for managing large code bases.

The reason for these limitations is that POSIX I/O and the dynamic loader are implemented as serial components of the operating system and do not take advantage of the parallel nature of the I/O operations. To avoid the above bottlenecks, this work describes two novel approaches for the integration of locality awareness (e.g., through aggregation or caching) into the serial I/O operations of parallel applications. The underlying methods are implemented in two tools, *SIONlib* and *Spindle*, which exploit the knowledge of application parallelism to coordinate access to file-system objects. In addition, the applied methods also use knowledge of the underlying I/O subsystem structure, the parallel file system configuration, and the network between HPC-system and I/O system to optimize application I/O. Both tools add layers between the parallel application and the POSIX-based standard interfaces of the operating system for I/O and dynamic loading, eliminating the need for modifying the underlying system software.

SIONlib is already applied in several applications, including PEPC, muphi, and MP2C, to implement efficient checkpointing. In addition, SIONlib is integrated in the performance-analysis tools Scalasca and Score-P to efficiently store and read trace data. Latest benchmarks on the Blue Gene/Q in Jülich demonstrate that SIONlib solves the metadata problem at large scale by running efficiently up to 1.8 million tasks while maintaining high I/O bandwidths of 60-80% of file-system peak with a negligible file-creation time. The scalability of Spindle could be demonstrated by running the Pynamic benchmark, a proxy benchmark for a real application, on a cluster of Lawrence Livermore National Laboratory at large scale. The results show that the startup of dynamically linked applications is now feasible on more than 15000 tasks, whereas the overhead of Spindle is nearly constantly low.

With SIONlib and Spindle, this work demonstrates how scalability of operating system components can be improved without modifying them and without changing the I/O patterns of applications. In this way, SIONlib and Spindle represent prototype implementations of functionality needed by next-generation runtime systems.

Zusammenfassung

Auf heutigen Supercomputer-Systemen belasten parallele Anwendungen, welche regelmäßig Checkpoints der im Hauptspeicher befindlichen Simulationsdaten erstellen, das Dateisystem enorm. Zum Beispiel werden auf Supercomputern mit einem verteilten Hauptspeicher die Checkpoints oft individuell von jedem Ausführungsprozess erzeugt, wodurch eine große Anzahl von Dateien entsteht. Neben der aufwendigen Handhabung der Dateien bewirkt dieses als tasklokaler I/O bezeichnete Zugriffsmuster zudem eine hohe Belastung der Dateisystem-Komponenten, die für die Verwaltung der Metadaten zuständig sind, was zu Verzögerungen im Programmablauf oder sogar zu dessen Abbruch führen kann. Ähnliche Auswirkungen durch die hohe Belastung des Metadatenmanagements findet man auch bei parallelen dynamisch gelinkten Programmen, die beim Start nach den benötigten Bibliotheken auf dem Dateisystem suchen und diese von dort laden.

Hauptursache der oben beschriebenen Verzögerungen ist, dass die für den I/O zuständigen seriellen Komponenten des Betriebssystems keine Vorteile aus der Parallelität der Anwendungen ziehen können. In dieser Arbeit werden zwei neuartige Lösungen vorgestellt, welche Charakteristika der I/O-Operationen von parallelen Programmen ausnutzen und mit geeigneten Mechanismen wie z.B. Aggregation oder Zwischenspeicherung die oben beschriebenen Engpässe vermeiden. Die zugrundeliegenden Methoden wurden in den beiden Werkzeugen SIONlib zur effizienten Speicherung von tasklokalen Daten und Spindle für das Laden von dynamisch gelinkten Programmen implementiert. Beide Tools nutzen verfügbare Informationen über die Parallelität der Anwendung, die Struktur der I/O-Komponenten und des Verbindungsnetzwerks des Supercomputers sowie die Konfiguration des parallelen Dateisystems zur Koordinierung und Optimierung der I/O-Operationen aus. Als eine Zwischenschicht zwischen der parallelen Anwendung und den vorhandenen POSIX-Schnittstellen können sie ohne Modifikation des Betriebssystems und mit minimaler bzw. ohne Änderung der Anwendung eingesetzt werden.

SIONlib wird bereits in Anwendungen für die effiziente Erstellung von Checkpoints eingesetzt und ist in die parallele Performance-Analysewerkzeuge Scalasca und Score-P für die Speicherung von Ereignisspuren integriert. Messungen auf dem Blue Gene/Q System in Jülich zeigen, dass SIONlib auch bei 1,8 Millionen Prozessen eine effiziente Ein-/Ausgabe von Daten mit bis zu 60-80% der nominellen Bandbreite des Dateisystems unterstützt, ohne Probleme beim Metadatenmanagement zu verursachen. Auch die Leistungsfähigkeit von Spindle konnte mit Hilfe von Benchmarks nachgewiesen werden. Zum Beispiel ermöglichte Spindle auf einem Supercomputer des Lawrence Livermore National Laboratory das gemeinsame dynamische Laden mit nahezu konstantem Zeitaufwand auf einer ohne Spindle erst gar nicht erreichbaren Größenordnung von über 15.000 Prozessoren.

Mit Hilfe von SIONlib und Spindle konnte in dieser Arbeit die Leistungsfähigkeit von Komponenten des Betriebssystems gesteigert werden, ohne diese oder die I/O-Muster der Anwendungen zu verändern. Damit stellen beide Werkzeuge Prototypen für die Implementierung von Funktionalitäten dar, die von Betriebssystemen der nächsten Generation bereitgestellt werden sollten.

Acknowledgment

I would like to thank Prof. Dr. Dr. Thomas Lippert, director of the Jülich Supercomputing Centre, for the opportunity to carry out my PhD project in this excellent research environment. Furthermore, I would like to thank Dr. Norbert Attig for supporting me to conduct my PhD research within his division Application Support at the Jülich Supercomputing Centre. Thanks to Dr. Rüdiger Esser who initially encouraged me to start my work towards the PhD.

This project would not have been possible without the support and guidance of my advisor Prof. Dr. Felix Wolf, to whom I owe deep gratitude. I also thank Prof. Dr. Uwe Naumann for serving as referee.

Furthermore, I am grateful to Dr. Bronis R. de Supinski, Dong H. Ahn, Matthew P. LeGendre, and Dr. Todd Gamblin for the collaboration on the design and development of Spindle and for hosting me at the Lawrence Livermore National Laboratory.

Thanks are due to my colleagues at the Jülich Supercomputing Centre for their help and advice on countless occasions. I would particularly like to acknowledge my colleagues Dr. Florian Janetzko and Dr. Kay Thust for their help to improve my thesis. Not to forget Dr. Daniel Becker, who helped me in the initial phase of my PhD research on SIONlib.

For discussion and support about the Scalasca integration of SIONlib I would like to thank Dr. Bernd Mohr, Dr. Brian Wylie, Dr. Markus Geimer and Christian Feld. I would also like to thank Dr. Michael Stephan and Jutta Docter for their technical support to run all the I/O tests on the Blue Gene/Q system JUQUEEN, and the administrators of the JUROPA cluster for supporting me to run Spindle there.

I gratefully acknowledge the computing time granted on the supercomputers JUQUEEN and JUROPA at Jülich Supercomputing Centre to perform I/O benchmarks for my research in this dissertation. Furthermore, I thank the Lawrence Livermore National Laboratory for supporting me to perform my I/O studies for Spindle on their Sierra cluster.

Last but not least, I deeply thank my wife Ulrike for her unwavering support and reliance during all the stressed situations in my PhD work.

Contents

List of Figures

1 Introduction

Simulation codes as a complementary pillar of scientific research have a high demand for computing power. As a result, programmers have to substantially optimize the application codes for supercomputing architectures. In general, the optimization focuses on parallelization in order to adapt the simulation program structure to the parallel architecture of today's supercomputers. In addition, the results of such simulations have to be stored persistently on disk to evaluate the data in post-processing steps or to feed it as input to subsequent simulations. Along with the parallelization of applications, also their I/O comes into focus as an additional topic of optimization. Especially, with the evolution of supercomputers in the recent years the computing performance increases much faster than the performance of I/O subsystems, which leads to an increasing portion of application runtime spent for I/O. Therefore, programmers have also to parallelize I/O in applications to benefit from the parallelism of the underlying hardware and infrastructure. Similarly, the start-up of parallel applications uses an I/O pattern that current operating systems do not support with parallel methods that exploit the parallel capabilities of the supercomputer.

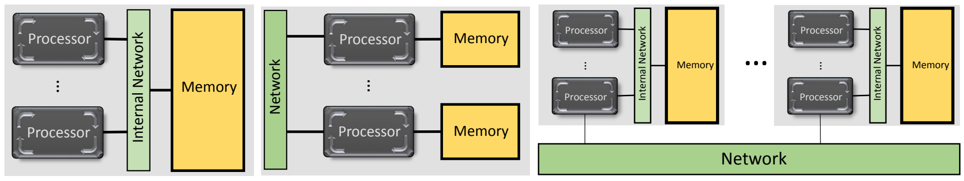
This chapter provides the basic concepts of parallel computing, parallel infrastructure, and parallel I/O as well as parallel task-local I/O and dynamic linking and loading of parallel applications. The latter two are needed to discuss the design and implementation of two new approaches for the integration of locality-aware methods into existing I/O layers of parallel applications. The major contribution of this thesis is the design and the implementation of the new tools *SIONlib* and *Spindle* to realize these approaches.

1.1 Parallel Computing

Parallel computing offers methods to run simulation programs at a much larger scale than traditional serial programs. The parallelism has to be implemented in computer hardware, system software and in simulation programs. For example, parallelism in hardware is given by duplicating computing elements on each scale. On small scale, this is implemented by multi-core CPUs or by using special accelerators like GPUs or Xeon Phi coprocessors; on larger scale, parallelism is given by connecting multiple compute nodes with an internal network. Furthermore, applications also have to be designed to run in parallel. The two main parallel programming paradigms supporting this are OpenMP [8] (*Open Multi-Processing*), designed for shared-memory systems, and MPI [68] (*Message Passing Interface*), designed for distributed-memory systems. We will discuss in the following sections the structure of parallel architectures as well as the two parallel programming paradigms with respect to I/O relevant implications, which have to be considered for the design of SIONlib and Spindle.

1.1.1 Parallel architectures

Modern supercomputers are constructed by combining multiple computing elements into a single system. As depicted in Figure 1.1 architectures of such supercomputers can be classified by their memory architecture as *shared-memory* systems, *distributed-memory* systems, or *hybrid-memory* systems.

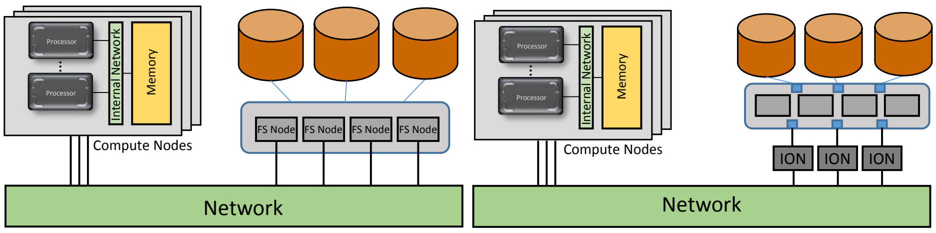


(a) Shared (b) Distributed (c) Hybrid

Figure 1.1: Architectures of parallel computers are classified by the memory configuration.

In shared-memory systems, all processors share a common physical memory and can access it via a shared *internal network*. The memory has a global address space, which is visible to all processors. Therefore, memory access has to be coordinated to avoid concurrent access to the same data. Systems where multiple processor units of the same kind are located on one board and which are connected to a shared memory are classified as *Symmetric Multi-Processor (SMP)* systems. This class also includes multi-core processors which combine multiple processor elements on one processor die (socket) and combinations of the two (multi-socket multi-core systems). Typically, SMP systems are equipped with a multi-level cache-hierarchy, consisting of processor-private and shared caches. Memory in multi-socket systems is often locally attached to a socket, so that processors have a direct access to data in the local memory bank and indirect access to data in more distant memory banks. The access to more distant memory must occur over the internal network and typically has higher latency and lower bandwidth. In contrast to the traditional design of SMP systems with *Uniform Memory Architectures (UMA)*, these modern SMP systems are classified as *Non-Uniform Memory Architectures (NUMA)*. In case of processor-local caches, the processors of NUMA systems have to ensure consistency of memory data in the different local caches. Typically, a hardware cache coherency protocol synchronizes the local caches. Architectures that implement such coherency are described as *cache-coherent Non-Uniform Memory Architectures (ccNUMA)*.

Computing elements of *distributed-memory* systems have a private local memory, which is only accessible to the local processor. In order to exchange data, processors have to transfer data explicitly from processor to processor and are thus interconnected via a network. Congestion of access to data in local memory is not possible since other processors have no direct access to it. Typical modern supercomputers are built as a combination of both distributed and shared memory architectures (cf. Figure 1.1c). Such systems consist of multiple SMP nodes, which are connected to a network. While memory is shared by all processors of the same node, it is not shared between processors in different nodes. Examples of such systems are massively *parallel multiprocessor (MPP)* systems like the IBM Blue Gene series or clusters like the Linux clusters JUROPA or Sierra, which were used in this thesis for I/O performance studies.



(a) Direct attached file system (b) External attached file system (I/O nodes)

Figure 1.2: Different schemes of file system attachment to a supercomputer.

Nodes are typically not equipped with local disks that can be used by applications. Instead, applications have to read and write data from a central file system, which is connected to the supercomputer via a network. Therefore, processors have to communicate directly with file-system nodes to read data from or write data to a file. Figure 1.2 shows two different configurations of how file systems could be connected to the parallel system. In the first configuration, the file-system nodes are coupled with the interconnect network of the supercomputer (cf. Figure 1.2a). File-transfer speed is then only limited by the speed of the interconnect network and their adapters to the nodes. Interconnect networks of center-wide file systems that are accessible by multiple supercomputers are often separated and in some cases implemented by a different technology (e.g. 10GE vs. Infiniband). Supercomputers in that configuration are connected via network switches or bridge nodes to the file system, which could limit the transfer speed to the file system further. The second configuration, shown in Figure 1.2b, describes the special configuration of the IBM Blue Gene system: Compute nodes have no direct connection to locations outside the internal torus network. Instead, nodes are connected via internal torus links to special I/O forwarding nodes (ION). On JUQUEEN, the IBM Blue Gene/Q system at the Jülich Supercomputing Centre (JSC), one ION is shared by 128 compute nodes. In this configuration, file data is transferred in multiple hops; each of these hops can further limit the file-transfer speed.

On distributed and hybrid-memory architectures there exists another implication for application I/O: data is distributed over the local memory of the compute nodes. To get access to all data, at least one processor per SMP node has to be involved in I/O operations. That means that applications are required to perform I/O in parallel on those architectures or to collect and distribute data on application level. In addition, distributed multi-dimensional and domain-decomposed application data has to be re-arranged before writing it to the disk. That requires I/O operations which map contiguous regions in local memory to non-contiguous data partitions in output files.

1.1.2 Parallel software

The parallel nature of the underlying architectures of modern supercomputers requires that applications also have a parallel structure. A parallel program is executed on multiple processors of multiple compute nodes, each executing one thread or task. According to Flynn's classification of parallel architectures [27], computers supporting such execution models follow the *multiple instruction multiple data (MIMD)* model: the application code is executed multiple times and typically, the threads or tasks are working on different sections of data. As is the case for the classification of parallel architectures, parallel programs can be classified into two categories, the *single program multiple data (SPMD)* and the *multiple program multiple data (MPMD)* category. Most of the applications on current supercomputers are following the SPMD scheme: an application that consists of one simulation code being executed on all processors. On the other hand, MPMD programs are special applications that implement multiple models (e.g. multi-physics applications) and therefore use a set of sub-codes. These sub-codes have to be executed concurrently on a subset of processors each. The MPMD programs are typically strongly coupled inside the sub-codes that are working on one model. The sub-codes themselves are more loosely coupled. They exchange data, for example, only after each or several simulation steps.

The behaviors of SPMD and MPMD programs are different in terms of parallel I/O and parallel loading: although threads of an SPMD program are working on different parts of the data sets, simulation data is typically domain-decomposed and an application will write output files that describe one (global) data set. In contrast, MPMD programs will typically output different data sets for each of the processor subsets, because they are simulating different sub-models. In terms of parallel loading, SPMD programs are running the same executable on each processor. Furthermore, dynamic libraries will usually be loaded in the same order. MPMD programs either will load a number of different executables, one for each subset, or will load the same driving main program but branch into different code segments afterwards.

To exploit parallel architectures, traditional sequential simulation programs have to be parallelized, either by message passing for distributed-memory systems or by multi-threading for shared-memory systems.

The Message Passing Interface (MPI) defines a standard applications programming interface (API) for programming distributed or hybrid-memory architectures. MPI provides language bindings for C and Fortran and implementations are available for nearly all parallel architectures. Therefore, most current applications, which are intended to run on such distributed-memory systems, are parallelized with MPI. Primarily, MPI defines, in its API, functions to exchange data between the processes of a parallel application. Although it is not described in the MPI standard, processes in an MPI application are typically called *tasks*. This document will adopt this notation and in the rest of the document this term is used to identify processes within a MPI context. Each task is assigned a unique number (*rank*), which makes it able to be addressed by other tasks. MPI tasks have their own address space and therefore have no direct memory access to data of other tasks. Although MPI is designed for distributed-memory systems, MPI applications can also run on SMP systems. In this case each MPI-task will run as its own Unix-process, which has its own address space.

As already described, MPI defines functions for data exchange. Data could be exchanged as messages between two tasks as a point-to-point operation or in collective operations between all or a number of application tasks (e.g. *broadcast*). To dedicate MPI operations to subsets of tasks, MPI provides functions to define *groups* of tasks. With the so-called *communicator*, a tasks group gets a communication context, which allows programmers to separate and limit communication to tasks of this group. Besides functions for data exchange, MPI provides a large number of additional functions (e.g. for task synchronization). Data, which will be sent to other tasks, have to be described to MPI. In addition to a memory pointer data exchange functions also require data type and size information. MPI defines a set of data types (*basic data types*) and provides functions to construct data types and data structures (*derived data types*).

MPI provides additional functions for parallel I/O with the MPI I/O API. This API uses knowledge about data structures for I/O optimization, which is given by the application through the type declaration of the data parameters. In addition, MPI I/O lets the applications describe how the data is arranged in the resulting output file (*file views*). This helps MPI, for example, to use in its library methods to reorder and aggregate data on the tasks before writing them to disk, which avoids non-contiguous file access. MPI scales to very large numbers of tasks: for example, codes from the High-Q Club [13] demonstrate their scaling to the full size of 1.8 million MPI tasks on the Blue Gene/Q system JUQUEEN at JSC.

Another class of programming interfaces supports shared-memory programming. The most popular programming interface is OpenMP, which supports a large number of platforms with parallel shared-memory architectures in its version 4.0. In contrast to message-passing models, OpenMP starts multiple light-weight threads within an application process and assigns these to the processors of an SMP node. An advantage of threading is that all threads have access to the same address space and can concurrently access data within that address space without needing to exchange data explicitly. OpenMP is a combination of compiler directives and library functions. Language bindings are available for C, C++, and Fortran. The structural parallelization of an application code is mainly done by inserting compiler directives. Starting from a sequential program, the user has to insert compiler directives to mark code parts that the threads can execute in parallel (*parallel region*). Program execution follows a *fork-join-model*, where the initial sequential execution is *forked* to multiple threads at the beginning of a parallel region, and where the parallel threads are *joined* at the end of parallel region.

Further OpenMP constructs are necessary to describe work sharing. Typically, computational loops are parallelized by assigning independent loop iterations to individual threads in a round-robin fashion. As the address space is accessible to all threads, OpenMP has to provide methods to organize concurrent access. This is done by declaring variables as thread-private or shared. Alternatively, parts of the code could be marked as *critical*, so that only one thread at a time will enter those regions. In comparison to MPI, OpenMP programmers have to spend most of their effort organizing data access, instead of coding the data transfer between application tasks with MPI functions.

On hybrid architectures, OpenMP will be often combined with MPI. In these configurations, MPI will run one or a few MPI tasks per SMP node and each MPI task is further parallelized with OpenMP. Depending on the thread-support level of the MPI implementation, MPI functions can be called concurrently from multiple threads within a parallel region (`MPI_THREAD_MULTIPLE`), serialized within a parallel region (`MPI_THREAD_SERIALIZED`), called only by the master thread (`MPI_THREAD_FUNNELED`), or called only outside of a parallel region (`MPI_THREAD_ SINGLE`). As already explained, most of the parallelization is done by defining work-share instructions for loops, which perform local computations. Therefore, MPI communication mostly occurs outside of OpenMP parallel regions, which requires the second lowest thread support (`MPI_THREAD_FUNNELED`) from the MPI implementation.

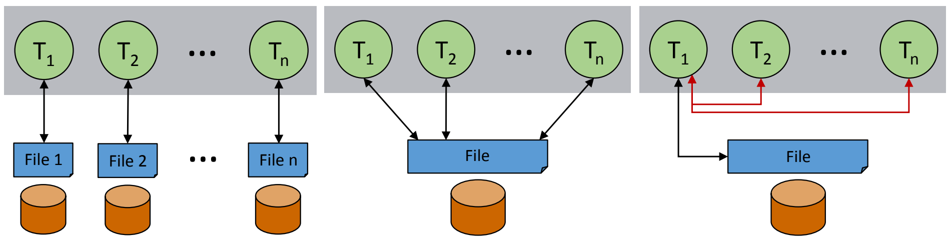
Similarly, application I/O is typically done outside parallel regions. Data which is intended to be written into output files is typically not thread-private. The master thread has access to the shared application data, so that this thread can write the total amount of output data sequentially to file. Although this strategy serializes the I/O on one SMP node, the parallelization of I/O is further possible on the outer MPI level by using APIs as MPI I/O.

Finally, the loading of shared-memory parallel or hybrid applications reduces the number of load operation at program startup, because fewer processes are started on an SMP node. This is because OpenMP threads are created by copying the execution context of the existing process and not by starting a new process. Therefore, the binary code does not need to be loaded again. Multi-threaded shared libraries are particularly useful, because the read-only section of libraries is loaded only once into memory.

1.2 Parallel I/O

Parallel I/O is defined by I/O operations that write or read shared data simultaneously from multiple tasks of a parallel application. Figure 1.3 depicts different approaches to implement parallel I/O [67]. A simple approach is to assign each task an individual file. Typically, this approach is used for writing checkpoint or restart files, where data belonging to individual tasks can be clearly separated. In this work, this I/O pattern will be referred to *parallel task-local I/O* (cf. Figure 1.3a). Alternatively, all tasks can access the same shared file, collectively or individually performing write and read operations on it (*shared I/O*). This parallel shared file I/O requires support on different software levels to coordinate concurrent access to the same location in a file from different tasks. I/O operations on shared files are typically supported by parallel I/O libraries, which use their functionality to manage the collective access to the shared file (cf. Figure 1.3b). A third approach, *centralized I/O*, is to limit I/O operations to one designated tasks. This task manages the write and read operations in a file on behalf of all other tasks (cf. Figure 1.3c). Furthermore, parallel applications and parallel I/O libraries often use combinations of this three approaches. This involves the use of multiple shared files where each is accessed by a subset of tasks.

The continuously growing number of processor elements of supercomputers and the growing size of usable memory, used by a single application makes parallel I/O for large-scale applications mandatory. However, the I/O capability of a single compute node is limited on current architecture which means the overall file-system speed can only be achieved by using all or a large number of compute nodes. Traditional approaches like collecting and writing data from



(a) Task-local I/O (b) Shared I/O (c) Centralized I/O

Figure 1.3: Different parallel I/O approaches, using individual files per task or a shared file.

only one instance of the parallel application would waste the I/O capabilities of the other nodes and is only reasonable for small datasets.

As a general introduction to parallel I/O, we will discuss different aspects from an application standpoint. The resulting characterization of I/O patterns and access methods helps to classify parallel I/O libraries according to the application interface, the integration into software layers, and the data representation in application and on disk. The parallel file system implements the low-level POSIX I/O interface, which is used by nearly all I/O libraries. Important aspects of the two parallel file systems used in this dissertation are introduced.

1.2.1 Characteristics

Different aspects of parallel I/O have a direct influence on the scalability. These aspects are the data file size, the frequency of I/O operations, the data distribution in distributed-memory systems, and the data representation in the memory and in the file. Additionally, how the data is stored on the file-system level (e.g. as shared or individual files) is an important factor for I/O scalability.

File size

Not all application data has to be exchanged between memory and the file systems. Some data is only needed for calculation and therefore only held in memory temporarily. Output results are computed or derived from the calculated data at the end of the simulation run or regularly after a number of simulation steps. Consequently, the size of the output file is typically smaller than the data held in memory. Therefore, the size of the output files containing the simulation results can be assumed to be smaller than the size of available memory.

The situation is different for applications that regularly store a snapshot of the current computation state to disk. This is required for restarting long-running simulations which have to be broken down into shorter jobs and for writing checkpoints, which allow restarting the computation after a failure. As such restart and checkpoint files contain a simulation snapshot, their size is typically of the same order of magnitude as the size of the application data in memory. File-system designers often consider such factors, when planning the capacity of a parallel file system: they must ensure that an application can write a certain share of the system main memory to a file in a reasonable time. Furthermore, as computing power and system main memory are growing faster than file-system performance, it will be more difficult to guarantee this for future systems. As a compromise, application programmers have to reduce the size of I/O data or reduce the frequency of writing checkpoints. In addition, new techniques like *burst buffers* may be used to address this problem by implementing faster local staging storage to compensate for longer writing times.

In contrast, input data of applications is typically very small. Depending on the simulation, only input parameters are read at the beginning of the simulation run. All intermediate data will be derived from these parameters. For example, the initial configuration of simulation data can be generated statistically from a set of parameters and boundary conditions. Another class of applications reads the initial simulation data from the output of a (parallel) pre-processing step. One example is the generation of irregular lattices for Computational Fluid Dynamics computations (CFD). Although these applications need high input bandwidth, most I/O infrastructures and I/O-libraries are optimized for data output. The reason is that the input phase happens only once per simulation run, whereas the data output and writing of checkpoints happen more frequently.

Frequency of operations

Next, we want to discuss the frequency and insularity of I/O operations in parallel applications. As already described, a program traditionally has an input phase during startup and an output phase at the end of the simulation run. In contrast, checkpointing is done regularly throughout the computation. Typically, checkpointing operations are completed immediately after the write operation. This means that checkpoint files are created just before and closed just after the write operation. This requirement is especially important for checkpointing, because operations have to be completed to ensure that the data can be used for restarting the simulation after a failure. There are also cases where applications write to a file continuously. Examples are log files, which contain information about simulation states. These files are opened at program start and closed at program end. This type of I/O can be denoted as *application monitoring*, which is needed to ensure that applications are running correctly. Monitoring is performed by the user by inspecting the log file regularly, which implies that write operations have to be completed in each simulation step and flushed to the file. This I/O driven application monitoring has its complementary counterpart in *online visualization* and *computational steering*, where data is directly sent to a visualization application instead of being written to the file system.

Data distribution and data representation

Simulation data on distributed-memory systems are typically distributed over the local memory of the compute nodes. The partitioning of data is mainly influenced by its dimensionality, the simulation algorithms, and the parallelization strategy. For example, three-dimensional data will be divided typically into sub-blocks, which are contiguous in the three-dimensional space. In contrast, memory and file space are one-dimensional (cf. Figure 1.4). Therefore, multi-dimensional data has to be folded into the linear space, typically for a three-dimensional field by iterating first the x-coordinate, then by iterating the y-coordinate, and finally by iterating the z-coordinate. Although this folding scheme keeps the data of sub-blocks in local memory in one contiguous block, it will not be contiguous in the folded representation of the global three-dimensional field. Instead, data of sub-blocks would be spread over different locations in the global representation. Because of this, and since applications typically store multi-dimensional fields in the folded global representation in output files, a reordering of data is necessary. Technically, this reordering can be done on different levels of the software stack. The following section, which describes the I/O libraries, will revisit this aspect.

As a compromise with respect to performance, applications often avoid the reordering of data stored in checkpointing- and restart-files. Instead, data is written as a sequence of sub-blocks into the output file. An application which wants to read such a file needs not only to know the field size in each dimension, but must also know the number and the size of each sub-block.

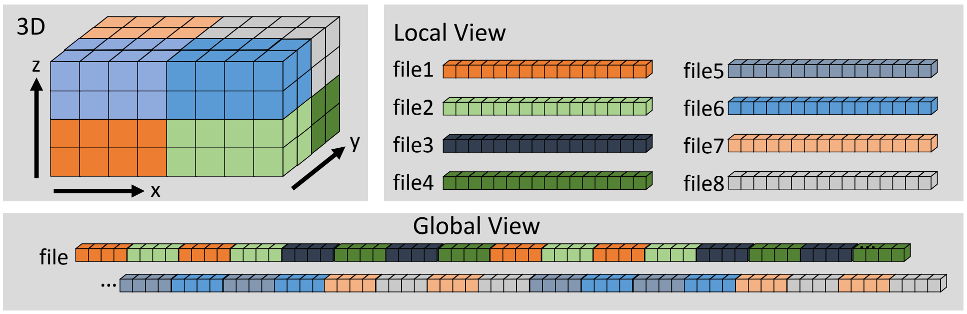


Figure 1.4: Three-dimensional domain-decomposed field stored in task-local or global file representation.

Restarting an application with the same domain decomposition from such a file is possible. However, starting the application with another number of tasks and therefore with another domain decomposition would require a reordering of the data during the input phase.

Data representation in file

Data is typically written as binary data to a file. Only in rare cases is data converted into human-readable ASCII format, such as log-files, which users can monitor directly. ASCII conversion requires additional time during writing and reading and is not necessary, if both writer and reader are part of a simulation program. However, writers of binary data need be aware of the binary representation when data is used on other systems: for example, binary data written on a *big-endian* system cannot be read on a *little-endian* system, because the byte-order of multi-byte data types (*words*) is different. Big-endian systems store the most significant bytes at the largest address, whereas little-endian systems store that bytes at the smallest address. Data conversion between both formats is only possible if the word-length is known.

Besides that, an application has to know the data representation in a file in order to be able to read it. In the simplest cases, data structure and data types are defined implicitly by means of direct coding in the corresponding read operations. The input file itself is only a stream of bytes and contains no additional information about data structures and data types. Such files are therefore dedicated to one application: they are not portable and cannot be used by other applications or tools using a different sequence of read operations.

Portability requires that data files be combined with *metadata*. Metadata describes the data format (e.g. binary, endianness), data types and structure, as well as the data size. The metadata can be coded directly into the output file or it can be stored in a different and possibly human-readable file.

More sophisticated file formats are those that store additional application-dependent metadata in the file. This includes both a description the structure and the meaning of the data (semantics). Examples for such self-describing formats are given in the next section with NetCDF and HDF5. Output files in this format can easily be used as input for a visualization program, which uses the metadata to present data sets graphically to the user.

Shared or individual file

With the increasing number of processors of supercomputers, how data is written to files has become more important. Writing data to individual local files yields a large number of output files, which have to be managed by the file system. This task-local approach is embarrassingly parallel. It does not require the synchronization of I/O operations in the application. Furthermore, with the one-to-one mapping of tasks and files, this approach can exploit all available data paths to the file system. Although this promises a good I/O performance, some operations, such as parallel file creation are very costly. Therefore, parallel I/O libraries are typically working with shared files, where all tasks of an application are writing to the same shared file, but at different positions. Parallel file systems are designed to handle such large shared files and have functionality to prevent data corruption when writing concurrently to a shared file (*locking*).

We have now introduced the following types of I/O: *program input*, *result output*, *restart files*, *checkpointing data*, and continuously written *log-files*. In the next section, we will discuss how such data can be managed with different I/O libraries and their access methods.

1.2.2 Access methods

Application I/O involves the interaction of the parallel program with the operating system and with the file system. To implement this I/O interaction, users can select from a large number of I/O libraries, which differ in terms of abstraction level and access method. Figure 1.5 shows a classification of representative I/O libraries and access methods, which we will discuss in the following sections.

Low-level I/O methods

On low level, the file system provides I/O functionality by supporting the standard of the *Portable Operating System Interface* (POSIX). The POSIX interface for I/O was designed in 1988 to have a common software layer between the file system, the operating system, and the application [47]. Initially designed as an interface for a single computer and its local file system, the POSIX I/O interface is currently supported by nearly all file systems. According to POSIX, data is stored in *files*. Additionally, these files have metadata, which describes their size, last access time, and access rights. Files are stored hierarchically in *directories* and are consistently visible to all clients accessing the file system. C-code programmers can directly use the POSIX I/O API while those using C++ and Fortran will need the support of language-specific modules (C++ streams, Fortran I/O).

With this tool set, applications can directly write data to files. A simple use case is parallel task-local I/O (cf. Section 1.3). One file object on the file system is assigned to each process. Operations on that file are done only by that process. Interaction or synchronization with other processes is therefore not needed. As described in the previous section, this approach promises a good I/O bandwidth, but has limited scalability due to the large number of files (cf. Section 2.2).

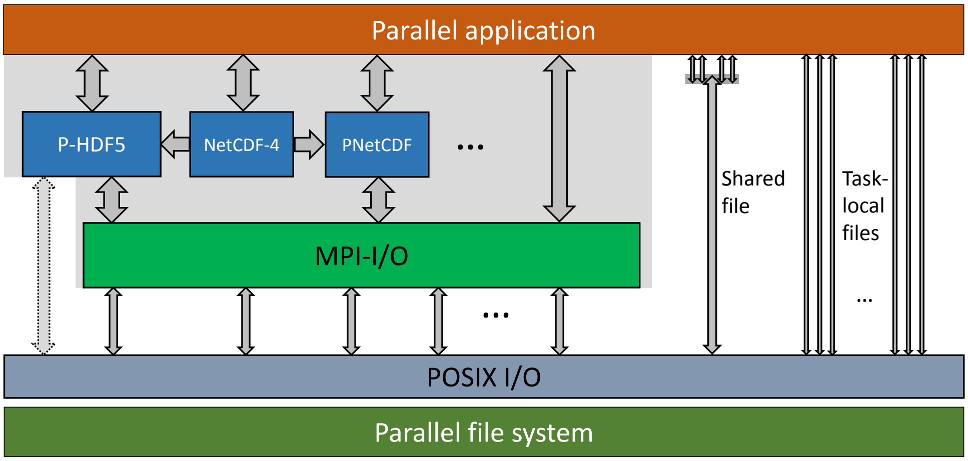


Figure 1.5: Classification of I/O libraries and their data access schemes.

Moreover, the POSIX I/O API can also be used to write collectively from multiple processes to a common shared file. The I/O operations are unchanged, but all processes open a common file instead of an individual one. To prevent conflicting access to the same byte ranges, the application has to coordinate access to the shared file. This can be done in a temporal distribution, meaning that access to the file is serialized among the processes, or in a spatial distribution, where the application will assign different non-overlapping chunks of the file to each of the processes. In case of concurrent access to a shared file, the parallel file system has to maintain precautions to prevent write operations at the same time to the same file location. For this, the file system typically implements a file locking mechanism, which grants only one process write access to the whole file or to a part of the file at a time.

The POSIX-based approaches for parallel I/O only store byte streams to files. The write and read commands of POSIX are very simple and accept as parameters only a memory pointer and size information. Data is written to the file as it is stored in memory. Information about endianness or other metadata have to be maintained by the application. The implementation of the POSIX I/O API is part of the standard system library *libc* and has no further dependencies on other libraries. Therefore, the implementation of application I/O with that interface is easy to use and popular.

MPI I/O

The main reason to implement special parallel I/O libraries is that the management of shared file access can be hidden in such libraries. Furthermore, parallel I/O libraries typically have access to the parallel environment of the application and can incorporate that knowledge into their design. The I/O libraries described in the rest of this section all follow this concept. Therefore, in an abstract view, these I/O libraries have to be positioned above the native POSIX I/O layer (e.g. MPI I/O in Figure 1.5).

MPI I/O as an extension of MPI implements two different strategies for parallel I/O. The first one is based directly on the POSIX I/O API calls and implements their corresponding counterparts in MPI syntax. Consequently, the write and read functions have an additional parameter that describes the MPI datatype of the payload. Those calls support I/O for both one-file-per-process and shared-file I/O, which were both described previously. Similar to the POSIX `seek` call, MPI implements a `seek` function or a combined `write_at/read_at` function, which allow processes to move to certain positions in the file. In addition, MPI provides a shared file pointer to coordinate the access to a shared file. With this shared file pointer, the MPI tasks collectively manage the current position in the file. For example, this feature is useful when writing into a common log file.

The second I/O strategy of MPI I/O requires more information about the data management in the application. It provides an additional set of high-level I/O calls: instead of writing chunks of bytes to a file, first the programmer has to describe data structure and the data distribution among the MPI-tasks with these I/O calls. With that information, MPI I/O is able to remap the distributed application data to the global view in the resulting file. This is done transparently using the write calls described below. MPI provides the *derived data types* and the *file views* for the data description. For example, the distribution structure of a three-dimensional domain-decomposed simulation field can be described directly with the `MPI_create_subarray` function and a corresponding `MPI_File_set_view` call (as depicted in Figure 1.4 on page 9).

MPI I/O write and read calls are implemented in two ways. The *independent* I/O operations can be called individually by a task without synchronization with other tasks and will be performed directly on the file. The second set of I/O calls is collective. They have to be called on all MPI tasks simultaneously, which offers more options for optimization to MPI I/O by applying collaborative I/O techniques. In this case, MPI can apply a two-phase I/O [23], where tasks rearrange their data in a first phase by exchanging parts of it such that in the second phase the data can be written to the disk with better performance. MPI I/O is therefore an example where the communication layer of the application (MPI) is used to optimize parallel I/O.

In addition, MPI I/O provides a separate function to pass *hints* to MPI I/O allowing the user to select between different I/O optimization schemes. In this way, it relays information about I/O access patterns and platform-specific optimization hints without modifying the I/O calls in the code.

High-level I/O libraries

On the next higher classification level of I/O libraries, we find the libraries P-HDF5, NetCDF-4, and Parallel NetCDF. In contrast to the previously described I/O-libraries, the underlying file formats are self-describing. Files of these formats contain not only the binary data, they also contain a description of the data. This description comprises information about data structure and type as well as a set of descriptive attributes. Importantly, these attributes can be used to make data interpretable for others. Overall, these formats are standardized, portable and, hence, can be used to archive data in long-term storage.

The Parallel NetCDF library [63] is the parallel access library for the NetCDF format, which is widely used in the climate community. Parallel NetCDF development was begun by Argonne National Laboratory in 2002 in order to efficiently create NetCDF files from parallel applications. The classic NetCDF file format is very simple. It allows the storage of set of variables and multi-dimensional arrays of fixed size as well as multi-dimensional arrays where one dimension is unlimited. Parallel NetCDF is based internally on the MPI I/O layer, which makes the previously described collaborative I/O techniques of the collective MPI I/O calls available in the Parallel NetCDF layer.

The HDF5 library [42] implements the rich *Hierarchical Data Format* (HDF) that allows the storage of data of different types and structures and supports hierarchical grouping of those entities. HDF files implement a kind of a file system within a container, using groups analogous to directories, and datasets similar to files [60]. The initial version of HDF was defined by the National Center for Supercomputing Applications (NCSA) in 1987. It became very popular in its latest format (HDF5), which was developed in collaboration with DOE laboratories and with support from NASA. Similar to Parallel NetCDF, the Parallel HDF5 library (PHDF5) implements a high-level parallel I/O interface, which is based internally on MPI I/O. In addition, HDF5 can also directly use the POSIX I/O interface for file access.

Besides these two previously described I/O-libraries, the initially serial NetCDF library has been enhanced in version 4 with a parallel implementation (NetCDF-4 [91]). Instead of interfacing directly with MPI I/O for file access, this new version uses either HDF5 to write NetCDF data in HDF5 format or Parallel NetCDF to write the classic NetCDF file format in parallel. In both cases, NetCDF-4 uses the parallel capabilities of other libraries to parallelize I/O operations, which are behind the native NetCDF interface calls.

High-level parallel I/O libraries like HDF5 and NetCDF provide I/O interfaces that require applications to provide information about the type and structure of data. They use this information to store the data in a global and self-describing format. Therefore, the focus of these libraries is on data portability and data preservation. In contrast, parallel I/O using low-level I/O-interfaces has other goals. In these cases, the data will stay on the system and is temporary. The focus of parallel I/O using low-level I/O-interfaces is often high I/O performance, where the data should be processed by the file system as quickly as possible. Typically, parallel task-local I/O is applied in those cases. As this is one of the main topics of this dissertation, Section 1.3 will describe this I/O type in more detail and we will discuss two use cases of parallel task-local I/O.

1.2.3 Parallel file systems

Parallel file systems are mandatory for large-scale parallel systems as they provide functionality for parallel access to shared file data from a very large number of computing elements. The parallel file system can be seen as a third component of a supercomputer. In large-scale systems, the file system itself consists of several server components, building together a parallel cluster with special support for I/O. In general, a parallel file system consists of two components: the hardware and software environment. Software is needed, on the server nodes, where file-system daemons provide access to the local file system for network attached clients. On the client side (e.g. compute nodes), software is also needed to interact with the remote file system. Depending on the file system, the software is either fully integrated into the Linux kernel (e.g. Lustre), or is partly running as an independent daemon (e.g. GPFS).

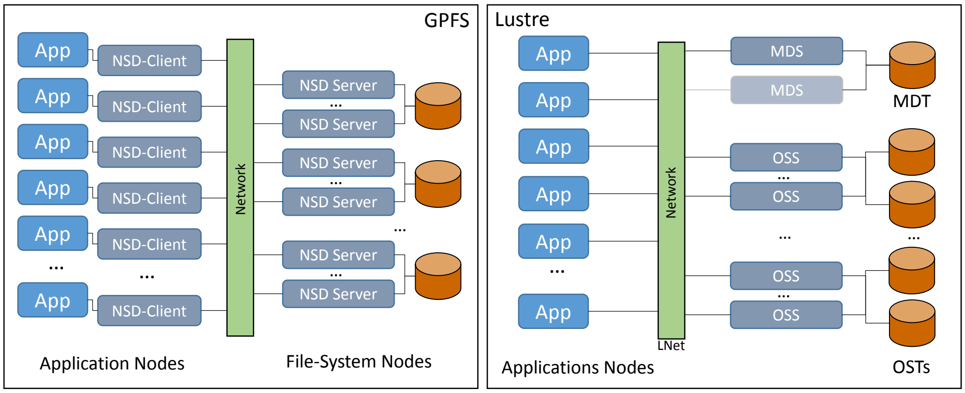
While HPC sites deploy a parallel file system for those data spaces that require high I/O bandwidth and support for parallel access, they are often using the *Network File System (NFS)* protocol to mount a center-wide HOME file system to the HPC system. While the NFS is not designed to support parallel access to files from a large number of clients, it allows a file system to be easily distributed to different computers of an HPC site. The HOME file system is typically used to store program source files and small configuration files. Parallel access is not necessarily needed for such a file system: compilation is typically done on the login-nodes of the HPC systems and job execution directories are typically located on the parallel file system. Storing binary files like executables or shared libraries on an NFS-mounted file system may result in bottlenecks and is therefore not allowed at some HPC sites. These sites require files to be copied to an execution directory on the parallel file system before job execution. Nevertheless, the limitations of loading shared libraries in parallel at large scale have motivated the design and implementation of a tool to support dynamic loading in this work (cf. Section 2.3.2).

Both SIONlib and Spindle, whose design and implementation are discussed in this dissertation, are approaches which were motivated by I/O limitations on HPC systems equipped with one of the two parallel file systems (GPFS and Lustre). While most of the optimization strategies will also be useful for other parallel file systems [76], the focus in the following description will be on these two aforementioned file systems.

GPFS

The *General Parallel File System (GPFS)* from IBM has its origins in 1993, when IBM started the development of the Tiger Shark file system. First commercial versions of GPFS were available under the name Parallel I/O File System (PIOFS). Since 1998, the file system has been distributed under the name GPFS [80, 44]. A large number of HPC systems listed in the current Top500 ranking list [70] are using GPFS to provide fast storage capabilities. The IBM Blue Gene/Q system JUQUEEN in Jülich is one of these. The GPFS file system of JUQUEEN is installed on a dedicated I/O cluster named JUST (Jülich storage server, [56]), which is an IBM System x GPFS Storage Server (GSS). It implements a new approach, where functionality of Redundant Array of Independent Disks, Level 6 (RAID-6) is not built into hardware controllers [21]. Instead, the RAID-6 functionality is performed by the GPFS native RAID feature (GNR) on the software level. Main reasons to move this functionality from hardware to software is that in case of a disk failure, the rebuilt speed is significantly improved.

As a cluster file system, GPFS provides access to file-system data from multiple nodes. Therefore, GPFS deploys a daemon on each node of the cluster. File systems are built on disk storage that is attached to one, some, or all of these cluster nodes. The file system is globally visible on all nodes and provides file storage with a global name space. Figure 1.6a shows a GPFS configuration using the *Network Shared Disk* (NSD) server model. In this configuration the GPFS daemons (NSD clients) that are running on the application nodes use the NSD block I/O protocol to communicate over the network to the GPFS daemons (NSD servers) that running on the file-system nodes. In general, GPFS daemons can take over file management tasks, which is a key element for scalability. For example, metadata of a file is handled by the first



(a) GPFS Network Shared Disk (NSD) Model (b) Lustre Architecture

Figure 1.6: General structure of GPFS and Lustre.

client that opens or creates a file. There are only a few file system tasks that are centralized and assigned to one GPFS daemon (e.g. the management of the file system itself).

Generally, parallel access to different files and concurrent access from different nodes to the same file is possible. To ensure protected access, GPFS provides a flexible and scalable byte-range locking mechanism. Initially, write or read locks for new or un-used files have to be requested from a central token manager. In contrast to other file system implementations, the lock tokens will then be delegated to the requesting daemon. Further daemons requesting locks for byte ranges of the same file must request a lock from the daemon holding the corresponding lock and will inherit a lock token for the requested byte-range. This mechanism offers a scalable parallel distribution of write/read locks in case of parallel access to a shared file.

In GPFS data is stored in units of file-system blocks. These blocks can be described as the smallest unit of data that can be handled efficiently. For example, GPFS daemons allocate an internal GPFS page pool in memory as buffer for incoming or outgoing data. GPFS pages are moved between memory and disk storage only as a whole block. Therefore, applications can achieve the best performance on the file system when they write and read data according to the file system block structure. This means that I/O requests have to be aligned to file system block borders and that chunk sizes should be a multiple of the block size. Efficient and easy-to-use I/O libraries should be designed to hide these constraints internally.

Lustre

The first version of the parallel file system *Lustre* [95] began development in 1999 under the name *object-based disk file system (obfds)*. The first version of Lustre [11] was deployed 2003 on a production cluster system at the Lawrence Livermore National Laboratory (LLNL). The development of Lustre has continued and the file system is installed on a large variety of large-scale HPC systems. Most of the HPC systems at LLNL are meanwhile driven with Lustre file systems, importantly, the Sierra cluster that was used for development and evaluation of Spindle, one of the tools in this dissertation, has access to one of the Lustre file systems.

Lustre is also used at the general-purpose cluster JUROPA at JSC. Two of the largest Lustre installations can be found at the Oakridge National Laboratory [84] and under the name FEFS (Fujitsu Exabyte File System) at the Riken Institute in Japan [79] where it serves as the file system for the K computer.

Lustre implements a distributed, object-based, storage. It internally handles objects to store data: files are stored in *data objects* and Lustre uses special *index objects* to store directory information and file metadata. The file system is implemented with *Object Storage Targets (OSTs)*, which are running on the *Object Storage Server (OSS)* nodes (cf. Figure 1.6b). The OSTs maintain local file systems to store the data, using *ext4* (*ldiskfs*) or other Linux file system implementations. Lustre builds the parallel Lustre file system by distributing file data over these OSTs. In contrast to GPFS, Lustre gives users and applications the right to influence this file distribution. To facilitate this, the OSTs of a Lustre file system are visible to users and each has a unique identifier. This allows users to select how a file should be distributed over the OSTs by using special `lfs` sub-commands before the file is created. The distribution model has only a few parameters: the number of OSTs, a starting offset number, and the chunk size can be specified. According to the specified parameters, the file byte range is divided into chunks which are distributed in a round-robin fashion over the selected OSTs during the write operation. Enabling users to select OSTs for their I/O operation could potentially cause an imbalance in OSTs usage. As we will see later in Section 4.3, such an imbalance can directly cause delays in parallel applications. However, the simple model does not allow for the selection of a set of OSTs by specifying their identifier, which limits the usability of the `lfs` sub-commands for implementing optimizations to prevent such imbalance.

The Lustre client is integrated into the Linux kernel. It can use the Linux file cache in memory to store Lustre file data. Therefore, Lustre need neither start an additional daemon on the client nodes nor allocate additional memory for caching. The Lustre kernel extension will communicate over the Lustre network abstraction layer (LNet) with the Lustre servers. In addition to object targets, Lustre implements *Meta Data Targets (MDTs)* on the *Meta Data Servers (MDSs)*, which are responsible for maintaining the name space of the file system. In contrast to the OSTs, only one metadata target is typically active for a file system; multiple MDTs are only used to build failover groups. The MDT stores the metadata information of files and directories (e.g. their names, access rights and ownerships). In addition, the MDS is responsible for selecting the OSTs that store file data according to the user specifications. The corresponding information about OST mapping and file distribution will be returned to those Lustre clients that open the file. After file creation, the MDS is no longer involved in the I/O operation. The Lustre clients directly interact with the assigned OSTs. The file metadata on MDS is updated after closing the file and the MDS is not involved in file block allocation. In contrast to GPFS, MDS functionality is not distributed over multiple servers. This causes bottlenecks when a large number of clients perform file look-up operations in parallel.

Lustre implements an additional service on each storage target for file locking, the *Lustre Distributed Lock Manager*. This is used to protect concurrent file and metadata operations, serializing such operations and ensures a consistent view. Similar to GPFS, Lustre implements byte-range locking. This enables shared-file I/O where different clients write disjointed chunks of a file.

1.3 Parallel Task-Local I/O

Historically, applications used often a one-file-per-task I/O access scheme to implement fast I/O for dumping large data as quickly as possible to disk. This scheme, which is described in this work as *parallel task-local I/O*, offers some advantages. The first is that contiguous data sections in memory are directly mapped to contiguous file data, which makes I/O operations very efficient. The second is that applications do not have to program additional logistics to align local I/O operations with other tasks; all I/O operations are local to the tasks and therefore independent.

The typical I/O interfaces for parallel task-local I/O are POSIX I/O and its equivalent language bindings like ANSI C I/O, C++ streams, or Fortran I/O. We will discuss the fundamental serial scheme of these I/O interfaces in the next section. Although the task-local file approach becomes a bottleneck at large scale, the fundamental scheme to preserve the contiguity of local data in output files persists as requirement of current applications. The following sections introduce two use cases which can benefits from an efficient implementation of parallel task-local I/O.

1.3.1 Serial I/O based on the POSIX I/O interface

As already described, POSIX I/O [47] is the native application interface of current parallel file systems. It standardizes access to file system data by providing a serial interface to write and read binary data. A subset of these functions are `open`, `close`, `read`, `write`, and `lseek`. A file will be opened with the `open` function. More precisely, this function connects the file with a file descriptor, which can then be used by other POSIX I/O calls to operate on that file. The file descriptor is of type `integer` and maps internally to a data structure that contains information needed by the POSIX I/O to operate on that file. For example, this data structure includes a file pointer for the current file position. Files can be opened for either writing or reading, which is determined by one of the function parameters. An additional flag allows the modification of the desired behavior of the `open` operation. This allows a file to be replaced or to have data added to it when opened for writing. The interface of the `write` and `read` function is quite simple: besides the file descriptor, which specifies the file on which the operation should be performed, these functions have a pointer and a number of bytes as parameters. Both functions operate without buffering data. Therefore, data is transferred directly between the current position in the file and the location in memory, which is specified by the pointer. The functions automatically advance the current file position by the number of bytes that are successfully written or read. In this way, I/O has stream characteristics: consecutive calls to the write or read function will operate sequentially on the file contents. However, the current position can be moved within a file using the `lseek` function, which implements a direct access scheme to a file.

In general, the POSIX I/O interface allows the same file to be opened from multiple processes at the same time. Developed for traditional local file systems, POSIX I/O has no special support to coordinate concurrent access. Furthermore, it defines semantics that guarantee the atomic behavior of write access. This requires that modifications to the file of such I/O calls have to be visible immediately to other processes [41]. Since a parallel file system has to provide a POSIX-I/O compliant interface, it has to guarantee these semantics. File systems often use more relaxed semantics, which guarantee consistency only after closing the file (e.g. NFS). Parallel I/O libraries that implement collaborative I/O operations among all tasks can arrange the underlying I/O operations in such a way that they guarantee the semantics without requiring this from the file system [43].

The C language defines an additional set of I/O functions, the ANSI C I/O stream interface [66]. Streams are used in the C language, for example, to implement formatted I/O to the standard I/O channels *stdout* and *stderr*. Furthermore, C also defines functions to open file streams (`fopen`) and to write (`fwrite`) and read (`fread`) unformatted binary data to/from files. Although the interface is different from POSIX I/O, its functionality is mainly the same. Moreover, since the file system handles I/O through the POSIX I/O interface, ANSI C has to implement I/O on top of the POSIX I/O interface. The main difference between the two implementations is that ANSI C implements buffered I/O by allocating an additional memory buffer for each file. This buffer is used for caching data of the I/O operations. The buffer size is adapted to the block size of the file system. This optimizes especially long I/O sequences of small data because data is aggregated in the buffer and the only I/O requests that arrive at the file system are those aligned to the file system block size. In contrast to this, applications that perform direct and random access to a file have to ensure that the data in the memory cache is consistent when moving the file pointer to another position in the file. Here, ANSI C provides the `fflush` routine for flushing the memory buffer. Another possible disadvantage of the memory buffer is its size. Typical file system block sizes are large on parallel file systems to ensure high I/O performance for large data I/O. For example, the *scratch* file system on JUQUEEN, which is intended to store efficiently large data, has a block size of 4 MiB. The memory requirement of an application can significantly grow with these additional buffers, especially when the application is simultaneously managing multiple files.

Similar to ANSI C, C++ implements a set of I/O streams in object-orientated classes which are typically buffered. The language bindings for the Fortran unformatted I/O differ, since the binary data representation in the output file has an internal record structure. Consequently, Fortran inserts additional descriptive data around the data of a write call. This allows Fortran to move easily from record to record in a file without reading all data. Both language bindings (for C++ and Fortran) are typically realized on top of the POSIX I/O layer.

1.3.2 Checkpointing with task-local I/O

In large-scale simulations application-level checkpointing is an indispensable technique for preventing data loss in case of system or program failures [17]. In this sense, applications have to regularly write the current state of the simulation to storage, from which the state can be recovered after a potential simulation abort. As explained in Section 1.2.1, the file size of checkpoint files can be in the order of the overall memory size of the application nodes. Due to the high demands that checkpointing imposes on the I/O infrastructure, application must use parallel I/O to benefit from the capabilities of a parallel file system. It is typical that checkpointing does not require a change in the local representation of simulation data in local memory to a global representation in the checkpoint file. The data represents the application state

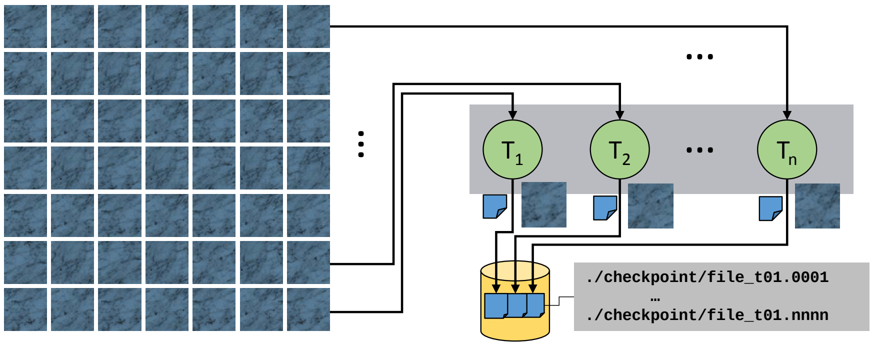


Figure 1.7: Example of checkpointing to task-local files from a parallel application. The I/O is embarrassingly parallel, because all tasks perform individual I/O operations.

at a certain simulation time, and it is intended only for restarting the application after a failure and for rolling back to the corresponding simulation state. The application state typically depends on the size of the simulation (e.g. number of tasks). In case of failure, the simulation is restarted with the same number of tasks. A conversion of data to a global representation, as depicted in Figure 1.4 on page 9, would lead only to an additional overhead of write and read time. Additionally, the probability of failure is very low on contemporary large-scale systems (e.g. BG/Q) . Therefore, checkpointing read operations are very seldom performed, and it is acceptable to rearrange the checkpointing data when a simulation is restarted with a different application size.

An important feature of checkpointing is that data describing a simulation state have to be dumped as quickly as possible into storage. As a consequence, application programmers often use parallel task-local I/O, which is easy to implement and promises a high I/O bandwidth – thanks to its embarrassingly parallel programming scheme. As illustrated in Figure 1.7, at a checkpoint the application creates a number of individual files, each containing the data from one task. Checkpoint files are often organized into directories, each containing the data from one checkpoint. Moreover, not all checkpoints are retained. A common and minimal scheme is to remove a previously written checkpoint after having successfully written the current one. That means that disk space for at least two checkpoints must be available. Schemes that are more sophisticated implement multi-level checkpointing, which manages different storage classes level to store checkpoint files: checkpoints are written at high frequency to fast, optionally local and less reliable storage. At lower frequency, checkpoint files are migrated to slower but more reliable storage systems. Examples of software implementing these schemes are SCR [73] and FTI [4]. The deployment of local storage has some implications. One of which is that file systems are local and do not provide a global name space. In this case, applications or underlying I/O libraries also have to manage the location of a checkpoint file as additional metadata. There are two ongoing projects at JSC which deploy local storage on HPC systems: in the EU project Deep-er [88] local storage will be integrated into the HPC file system structure with the help of the Fraunhofer parallel file system BeeGFS [28]. In the other project, local storage is deployed on the Blue Gene/Q system JUQUEEN at JSC. This *Blue Gene Active Storage* (BGAS) not only provides fast local storage to Blue Gene I/O nodes, it also provides local computing capability on the I/O nodes to work actively on locally stored data [18]. The optimization techniques for parallel task-local I/O, which are described in in this dissertation, are to be tested and verified in both projects.

A positive side effect of checkpointing is that the checkpoint files can also be used as regular restart files of applications, particularly if the simulation has to be performed in multiple subsequent compute jobs because the overall computation time is longer than the maximum wall clock limit of compute jobs on an HPC system. For example, jobs on JUQUEEN are limited to 24 hours. Long-running simulations have to be split into multiple 24h-jobs, whereas each subsequent job has to be started with the last system state of the previous job. Therefore, in this case, restart files are kept after the successful end of a compute job, in contrast to checkpoint files.

An example of checkpointing in an application will be discussed in Section 4.4.1, In this section the application state of MP2C, a simulation code for Massively Parallel Multi-Particle Collision Dynamics, is mainly defined by positions and velocities of the simulated particles. The overall number of particles can grow in large simulations with this code on the order of billions, which leads to multi-TiB checkpoint files.

1.3.3 Performance-tools using task-local I/O

The second use case for parallel task-local I/O is not directly driven by application requirements: besides application checkpointing, high scalable tools may also deal with task-local data. As an example, we will discuss the I/O requirements of the Scalasca toolset [36]. The open-source toolset Scalasca analyzes parallel applications in order to find possible performance limitations and to identify opportunities for optimization. Scalasca can work in two different measurement modes. In the profiling measurement, Scalasca collects performance information during runtime, this includes execution times of application functions and the time spent in calculation or communication. In this way, it provides a performance profile of the program execution. In measurement mode, Scalasca records event traces during runtime and analyses these afterwards in a separate step with a parallel trace analyzer program. Events are captured when a process enters or exits a program function or a code region or when communication with other processes is triggered. The event trace measurement allows the identification of wait states in a program that occur as a result of unevenly distributed workloads. Furthermore, this measurement also allows the identification of critical paths and root causes of such wait states automatically [9]. The event traces are collected on each process and have to be stored on disk in temporary files. Due to a potentially high number of events, this measurement can generate a high I/O load during the measurement and the analyzer step.

Technically, Scalasca instruments the parallel application and stores the recorded events in a memory-based collection buffer during runtime. The content of these task-local buffers is written at the end of the measurement to task-local files. As memory is limited, the collection buffer has a limited size and can only store a certain number of event records. However, the overall number of events that have to be recorded on one task is not limited. Therefore, Scalasca has implemented so-called *intermediate flushes*, which empty the buffers directly to file as soon as they are full. This increases the complexity of the I/O pattern as write operations can happen randomly and are not coordinated among the tasks during runtime. Following the

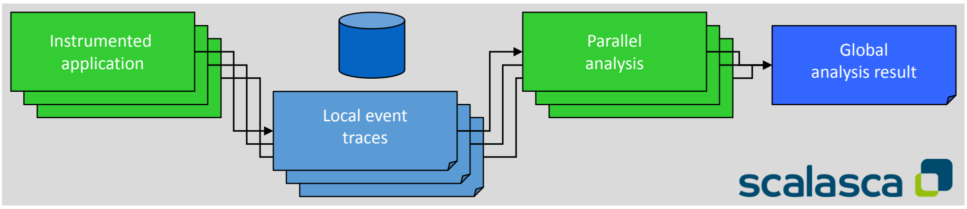


Figure 1.8: The parallel trace analysis in Scalasca consists of two steps. The first step runs the application and creates a local event trace file for each application task or thread. These files are read again in the second step into the parallel trace analyzer tool.

work flow depicted in Figure 1.8, the traces are loaded postmortem into the distributed memory of a parallel trace analyzer program. The trace analysis is available for MPI applications and for the hybrid MPI/OpenMP programming model. Similar to MPI tasks, OpenMP threads perform the event trace collection individually. Therefore, task-local trace files are generated for each OpenMP thread. In general, the number of these trace files is not limited and not known in advance as OpenMP threads can be generated dynamically during runtime (e.g. OpenMP nesting). However, Scalasca assumes a fix number of OpenMP threads per task. Although the completion of trace analyses for applications running on up to 64 K cores has already been demonstrated [36, 72], a notable bottleneck was found in the initial version of Scalasca during the experiment activation (i.e., creating the trace files and initializing the tracing library), as one would expect.

The newest generation of Scalasca (2.x) has been structurally changed [96]: the components, which instrument the parallel application and collect the event traces, are functionally replaced by components of the Score-P instrumentation and measurement infrastructure [59]. Similar to the first version of Scalasca, Score-P produces trace data for each task or thread of a parallel application. The data will be stored in the *Open Trace Format 2 (OTF2)*, where the Score-P infrastructure uses an abstract I/O layer (*substrates*) for the I/O operations. Score-P supports not only Scalasca, but also works with the performance tools Periscope [37], Vampir [58] and Tau [83].

1.4 Dynamic Linking and Loading

Dynamic linking has many advantages for managing large code bases. Splitting a monolithic executable into many dynamically shared object files (i) decreases the compile time for large codes, (ii) reduces runtime memory requirements by allowing modules to be loaded and unloaded as needed, and (iii) allows common libraries to be shared among many executables. Dynamic libraries are often used in small and mid-size applications, such as software for desktop systems. However, software complexity is increasing in high performance computing (HPC) applications and dynamic linking and loading [62] also offers advantages for managing this complexity.

Dynamic libraries allow large applications to be modularized, or split into independently built packages. Modules can greatly reduce build time for developers, and each module can be loaded and unloaded dynamically at runtime as needed. This saves memory and frees developers from the burden of maintaining multiple pre-configured builds of large applications. Furthermore, dynamic loading is used extensively by dynamic languages such as Python. High-level abstractions in these languages enable the rapid development of new plugins and packages, and they reduce software complexity for application programmers. These features are increasingly used to manage complexity in large, multi-physics simulation codes, which may have millions of lines of code and thousands of runtime configurable parameters.

Dynamic linking and loading first appeared in the MULTICS [22] operating system (OS) in 1964 as a way to share common code between processes. For this reason, dynamically linked libraries are often called *shared libraries*, and any object that can be linked and loaded by the dynamic loader is called a *Dynamic Shared Object (DSO)*.

1.4.1 Dynamic shared objects

DSOs are stand-alone containers for compiled code and program data. DSOs are linked and loaded at start-up time or at runtime to the program. The data format for storing DSOs and executable files on Unix-like systems is the *Executable and Linkable Format (ELF)* [20, 90]. DSOs are independent objects and have to be able to be placed at any desired position in the address space of a process (relocatable). This means that absolute references to memory positions of library symbols have to be re-computed after the object is placed into memory. To avoid this, a compiler can directly create *Position-Independent Code (PIC)*, which uses only relative memory address references. With this independence from a location in address space, DSOs can be linked against different executables without re-compilation. In addition, running multiple instances of a dynamically linked program on a shared-memory system avoids multiple instances of the dependent DSOs in memory. This is possible because the read-only parts of the DSO (e.g. code section) can be shared.

ELF objects are structured in sections that describe different code components (cf. Figure 1.9). Sections are available, for example, to store code, data, or symbol tables. The symbol table

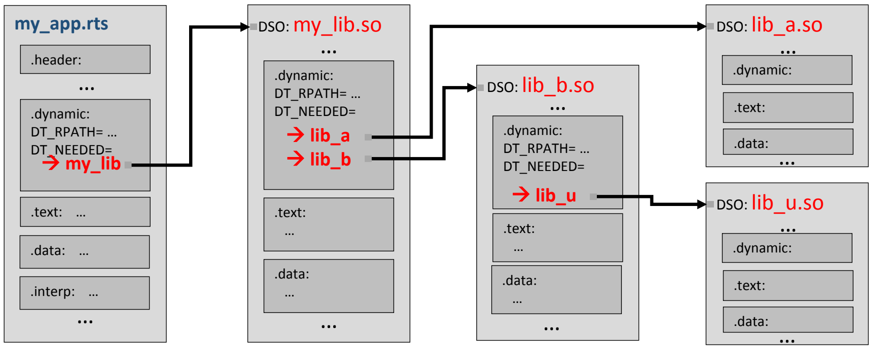


Figure 1.9: DSO structure and dependencies. A DSO in ELF are structured in different sections. References to code in other libraries lead to dependencies between DSOs, which are stored in the `.dynamic` section.

contains references to function or data objects which are accessible from outside the DSO. The dynamic loader will use these tables to combine code sections from different DSOs to create executable code.

Libraries can contain references to codes from other libraries. These libraries have to be loaded and added to the executable in the address space in order to resolve all references. The library names are stored in the attribute `DT_NEEDED` of the section `.dynamic`. This section also stores a number of other attributes which are required for dynamic loading. One example is the `DT_RPATH` attribute, which contains search path information for locating the related libraries.

1.4.2 The dynamic loader

The dynamic loader, also called the dynamic linker, is responsible for locating object code and making it available within a process's address space. These actions make the object code's subroutines callable by the main program. Most HPC systems, including Linux clusters, IBM Blue Gene, and Cray machines, use the dynamic loader implementation from GNU's libc. This loader is based on standards detailed in the System V Application Binary Interface (ABI) [90].

The dynamic loader is implemented as a DSO that is loaded by the OS during process start-up. A reference to the loader is stored in the `interp`-section of the dynamically linked application. At startup time the OS examines this section and gives control to the specified loader, which then loads the main executable's dependent libraries and transfers control to the executable's entry point. Figure 1.10 depicts this procedure: the dynamic loader parses the dynamic section of the dynamically linked executable for `DT_NEEDED` entries. For each of these entries the loader will search within a set of locations for a library file with a specified file name. When found the file will be read into memory and inserted into the address space of the process. This procedure will be repeated for each entry in this DSO and in subsequently loaded DSOs.

During the load procedure, the dynamic loader performs two types of file-system operations: query operations to locate a DSO and read operations to load its contents into memory. Locating a DSO is necessary because an executable does not specify its dependent libraries with full path information. Instead, the executable provides only the *names* of the required libraries. To load a particular library, the dynamic loader searches for files by name in system locations (e.g., */lib*), directories named in the executable (rpaths), or directories named in environment

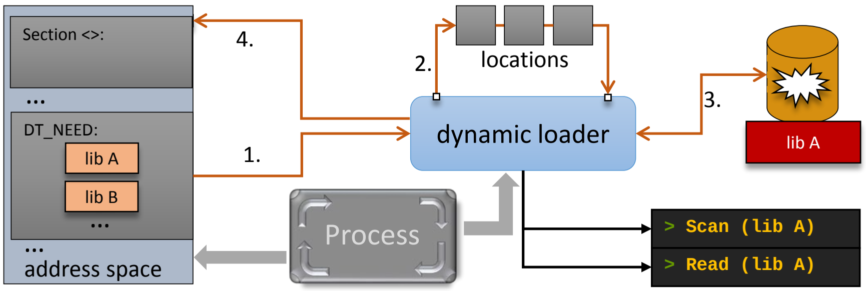


Figure 1.10: Load process of dynamically linked executables. During startup of the executables, the dynamic loader searches in a list of locations for the required libraries.

variables such as `LD_LIBRARY_PATH`. The GNU implementation tests for existence by appending the name to each directory and calling the POSIX `open` function for the resulting path. This look-up protocol is not scalable because the search operations are repeated for each library and for each process of a parallel application.

The second type of file-system operation is loading the library file into memory. Similar to the first type of interaction, POSIX I/O calls are used to perform the read operation. Once a library has been located, the dynamic loader maps it into memory. Each library contains a table of program headers that describe which parts of the library on disk should be mapped into memory. Typically, its code and data are mapped into memory while its debug and linking information are not. Unfortunately, the loading protocol is – similar to the look-up protocol – not scalable. The GNU loader uses `open` and `read` system calls to access the program headers, then the `mmap` system call to load the bulk of the library into memory.

The executable may re-invoke routines in the dynamic loader to resolve symbols or to load new DSOs during runtime (via routines such as `dlopen`). Typical use cases are applications that have a script-based driver environment. Section 1.4.5 will discuss this use case in more detail.

1.4.3 Optimization of symbol binding

Position-independent DSOs (e.g. PIC) have more flexibility in terms of re-use. The downside of this characteristic is that symbol binding has to be done by the dynamic loader at runtime and not the dynamic linker at compile time. Binding of symbols is required to ensure that references to external objects are resolved. Examples for such external references are calls to functions that are stored in other DSOs. The symbol binding has to be done at each program loading and in a parallel application within each parallel process. On the other hand, symbols in statically linked programs are bound only once at link time.

Before binding a symbol to a memory address, the loader has to look up the symbol in special tables, which are generated during the load process from information of already loaded DSOs. Performing this operation for each symbol would be too costly, especially since in typical programs only a part of the functions of a DSO are referenced. The ELF format and the GNU loader implement for this *lazy binding* as an optimization of the load process (described in [5]): symbols are resolved the first time they are referenced at runtime.

Figure 1.11 illustrates the implementation of the technique with the help of two additional sections in the DSO: the *Procedure Linkage Table (PLT)* and the *Global Offset Table (GOT)*. The first data structure, the PLT, is a list of short code fragments that are needed to perform lazy binding. Calls to external functions are directed to these entries. The first instruction of the PLT entry is to jump directly to an address that is stored in the corresponding entry of the second data structure, the GOT, which contains absolute addresses of external functions in the process address space. At initialization time, the external symbols are not resolved. Instead, the entries in the GOT contain an address that points back to the second instruction of the corresponding code fragment in the PLT. The following code of the fragment executes a look-up procedure for the symbol and replaces the GOT entry with the real address of the called function in the process address space. After this lazy binding, following calls to an external

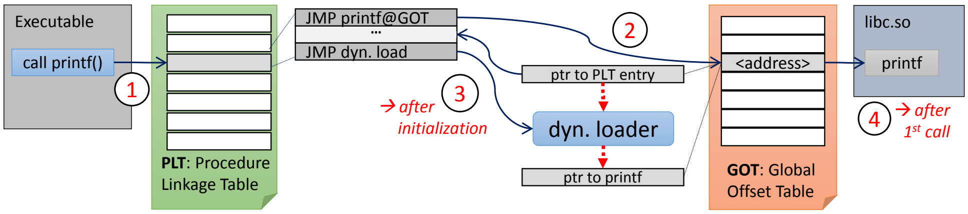


Figure 1.11: Lazy binding optimization of dynamically loaded code. The symbol location in process address space is resolved the first time the process execution reaches references to that symbol in the code.

function will jump directly to the real address of the function. The look-up procedure is part of the dynamic loader and will be called only once for each symbol with this method.

The lazy binding optimization reduces the number of look-up requests to the number of function calls that are really needed. However, the look-up requests are shifted from initialization to process execution and can therefore produce disturbing noise in the application. Overall, this optimization is needed for efficient execution of dynamically linked programs, especially for parallel applications. The GNU linker enables the lazy binding technique by default.

It is remarkable that only the global offset table has to be stored in a segment with read/write access rights. The code section and the PLT do not change during this load process and can therefore be stored in a read-only segment.

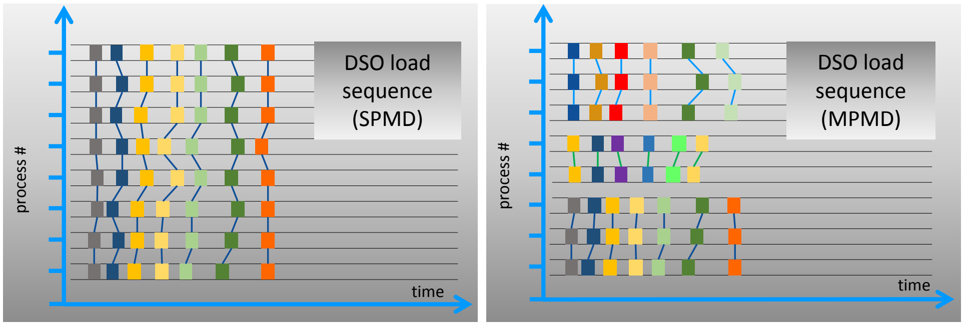
1.4.4 Dynamic linking and loading in parallel applications

Processes of parallel applications, which follow the SPMD scheme, execute the same code on each process and only one executable is used. As the required DSOs are listed as DT_NEEDED entries in the dynamic section of the same executable, the sequence of load requests of the processes is identical. This results in identical sets of DSOs as depicted in Figure 1.12a.

Large simulation codes sometimes do not follow a pure SPMD scheme. Instead, they consist of multiple partitions, each dedicated to one part of the simulation. The simulation of coupled models is an example of a setup that follows the MPMD scheme. The number of partitions is small compared to the number of processes in the parallel application and because multiple processes are running the same executable, the sequence of load requests and the set of DSOs are identical. However, processes running different executables may have different sets of DSOs and load orders of DSOs, as demonstrated in Figure 1.12b.

The load sequences of DSOs are not only similar for all processes; they are also concentrated to a short time interval: most load requests occur at program start-up. The load operations are performed concurrently on all tasks, but they are not synchronized. Because of the missing synchronization, the start time of library load operations can vary among the processes as indicated in both cases in Figure 1.12.

The dynamic loader is not aware of the parallel nature of the underlying application. The loader operates in serial for each process and cannot take advantage of the parallel nature of



(a) SPMD (b) MPMD

Figure 1.12: Library load sequences of parallel programs. Setups in SPMD scheme issue library loads in the same order on all tasks. Setups in MPMD schemes typically consist of multiple SPMD blocks that each locally loads the same set of libraries in the same order.

the application. Therefore, each task will execute the previously described search and look-up operations in the same execution order in parallel. This disadvantage will be discussed in more detail in Section 2.3.1.

1.4.5 Dynamic loading at runtime and within scripting languages

As described in the previous sections, the dynamic loader supports dynamically linked programs where DSO dependencies are evaluated and where DSOs are loaded at program start. Furthermore, the dynamic loader also supports library loading at runtime. To interact with the dynamic loader from a user program, the application programmer has to insert special system calls like `dlopen`. The user program passes a library name and load options to the `dlopen` call. The function returns a *handle* for the specified library. Afterwards, functions of the library can be called directly via this library handle. Applications use this feature in order to determine which auxiliary library has to be loaded at runtime according to the configuration options in the input file.

Furthermore, dynamic loading at runtime is also used extensively by dynamic languages such as Python. Scripts in interpreted languages run often as driver scripts of applications as these are easier to manage and to configure than input files of compiled programs. Computation intensive parts of the application are typically written in languages like C or C++, stored in external core libraries, and included at runtime via `dlopen`. Script-driven applications allow easy configuration and programming of application workflows in combination with efficiently implemented compute kernels in core libraries.

In script environments like Python, it is characteristic that not only libraries are searched and loaded during runtime. In Python libraries are integrated within functional modules, whereas files containing the byte-compiled Python code of these modules also have to be searched and read. Read operation of such files can occur more often than read operations of shared libraries. Similar to the dynamic loader, the files are read from each Python driver process individually.

The characteristic that load sequences of libraries are similar on all processes and are concentrated to short time interval can only be assumed at program startup and not during runtime. For example, `dlopen` may only be used on a subset of application processes and these processes can load different sets of libraries. These choices are determined by input configurations or intermediate results of computation. The load operations may also not concentrated on a short time interval because even if all processes load the same DSOs via `dlopen`, the load operations may not start at the same time or in the same order.

1.5 Contribution of this Thesis

Common use cases of parallel task-local I/O are application checkpointing and the non-volatile storage of temporary data in tools and applications during the runtime of a job. Both require a high I/O bandwidth from the parallel file system, because the size of the data is often on the order of the size of the available memory. Particularly at large scales, the performance of traditional parallel task-local I/O is limited. As we will discuss in Chapter 2, this limitation is mainly caused by metadata handling which can result in the creation of tens of thousands of files in a directory. Bottlenecks can originate from the serial nature of the POSIX I/O interface, which ignores the parallelism of the I/O operation and passes the burden of metadata handling onto the file system. Additionally, POSIX I/O is ultimately a limiting factor when starting parallel applications, which are dynamically linked and which add additional binary library data to the process during startup or runtime. The dynamic loader is designed as a serial tool and uses POSIX I/O internally for file access. Therefore, in this way it does not exploit the parallelism of the loaded applications.

POSIX I/O, is provided by the file system as the general interface to interact with the operating systems and applications. The interface has a long history. Enhancements to this interface require a long time to be implemented and manifested. However, the increasing size of parallel applications and the underlying parallel HPC systems, requires a modernization of the POSIX I/O interface, at least for the previously described use cases. The main goals of such an enhancement are (i) to provide high I/O bandwidths at large scale and (ii) to simultaneously limit the metadata management overhead for applications and tools, which perform task-local I/O operations at large scale. To accomplish these goals, a solution should exploit application and HPC parallelism for task-local I/O operations. Such a solution should operate in user-space without modification of the runtime and file system. The solution should leave the task-local file I/O strategy unmodified and it should be scalable, both in terms of I/O bandwidth and metadata overhead.

This dissertation introduces two new approaches that fulfill the above-mentioned objectives. These approaches are implemented in the parallel I/O library SIONlib and with the tool Spindle, which supports the efficient loading of dynamically linked executables at large scale.

The main issue of traditional parallel task-local I/O is the generation of a large number of individual files, which causes overhead in file creation and management. The first major contribution of this dissertation, the parallel I/O Library SIONlib, replaces the individual files by a small number of shared files preventing such overhead. Application tasks are assigned one or more chunks of these shared files, to which the task has exclusive access. This approach leaves the task-local file I/O strategy of applications unchanged. Furthermore, SIONlib uses the communication layer of the application to aggregate and distribute metadata among the tasks, which guarantees scalability to the size of the simulation. Efficiency is further improved by exploiting the I/O infrastructure and file-system properties, for example, to align data or to organize I/O aggregation according to the hierarchy of the I/O infrastructure. However, metadata performance in shared file I/O also suffers at very large scale (more than 64k tasks). Therefore, SIONlib can also use multiple files to build a virtual shared file. This strategy parallelizes metadata handling, because the physical files are handled by different components of the file system. As a result, the techniques, which are presented in this dissertation and implemented into SIONlib, enable applications to perform parallel task-local I/O on shared files with comparable I/O bandwidths and considerably less metadata overhead. For example, as described in Section 4.4.1, SIONlib improves I/O efficiency of the application MP2C. Now it can scale to the full BG/Q JUQUEEN system, running with 1.8 million tasks and writing checkpoint files of several TiB with more than 50 % of file-system peak.

The second major contribution is the efficient support of the dynamic loading of parallel applications. The approach, which will be presented in this thesis, consists of three main techniques, (i) the interception of the dynamic loader, (ii) the distributed caching of load-path information and the library data, and (iii) the deployment of an overlay network to connect the distributed cache servers. All these components are running in user space. Modifications to the runtime system and to the applications are not necessary. For example, the standard GNU Linux loader provides a user-space interception technique, which allows Spindle to monitor library load requests and to modify the results of look-up operations. The distributed cache buffers the results of library look-up operations and thus prevents multiple similar accesses to the file system. In addition, with the help of the overlay network library data is transferred to a location near the process that needs it. Taken these techniques reduce the number of file-system operations for loading a parallel, dynamically linked application to those needed to load a single program instance; the dynamic loading overhead remains constant.

Both, SIONlib and Spindle represent transient solutions that implement approaches which even demonstrate their efficiency at scale. They should, therefore, be adopted in the next generation of large-scale runtime and file systems.

The dissertation is organized as follows. After the introduction of both, parallel task-local I/O and dynamic loading in Chapter 1, Chapter 2 discusses the limitations of these techniques at large scale and the root causes of these limitations. Chapter 3 presents the design of SIONlib, which implements the approach of improved parallel task-local I/O. Next, Chapter 4 shows the results of evaluating these techniques with artificial I/O workloads as well as with real-world applications. Chapter 5 presents the design of Spindle, an approach designed to aggressively cache library look-up and load operations. Spindle performance is then evaluated on two different platforms in Chapter 6. Finally, Chapter 7 gives a concluding summary of the dissertation and provides an outlook for future research based on both SIONlib and Spindle.

2 I/O Limitations at Large Scale

With the increasing number of tasks of parallel applications, the efficiency of application I/O becomes more crucial. Similar to the parallelization of the computational part of the application with programming models like MPI and OpenMP, the I/O of the application has also to be parallelized. Common tools for this are, parallel I/O libraries like MPI I/O, pNetCDF, or parallel HDF5 on a higher, application oriented abstraction level, whereas task-local I/O patterns are typically realized with native POSIX I/O operations on a lower file-system oriented abstraction level (cf. Figure 1.5 on page 11). As already described, this work focuses on task-local I/O patterns that are given by I/O operations to individual files of parallel applications and dynamic loading of library data for starting parallel applications. Both patterns have scalability issues at larger scale. This chapter discusses the limiting factors, which hinder applications with such I/O patterns to scale to very large number of tasks. The limitations are addressed by I/O strategies that are designed in this work and which are implemented in two tools SIONlib and Spindle. In the following, an abstract model for parallel I/O data flow on complex I/O infrastructures is introduced, which helps to explain the limitation of the different types of parallel task-local I/O and dynamic loading patterns.

2.1 Schematic View of the Parallel I/O Data Flow

The objective of I/O is to move data between different storage resources. In the simplest case, data is moved by an I/O call from a location in the local memory of a compute node to a file on a local disk. In systems, which are more complex and support I/O from multiple compute nodes, the data will be moved in a number of steps from a compute node's memory to a disk of a shared parallel file system.

Figure 2.1 shows a simple model of an abstract I/O node, which represents one processing step in the data flow from compute node to the file system. In addition to the input and output

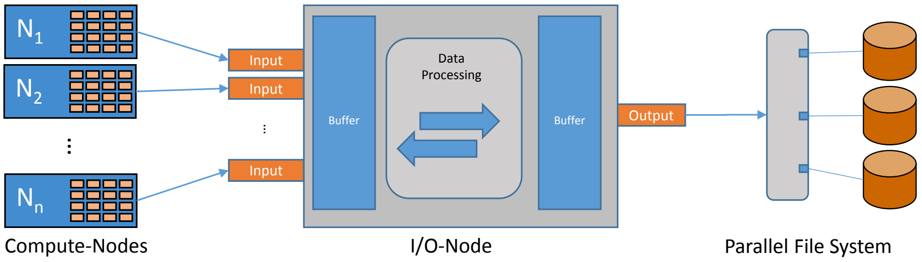


Figure 2.1: Abstract model of a simple I/O infrastructure.

streams, an abstract I/O node has also memory buffers for incoming and outgoing data. With the use of these buffers, I/O-operations of the application can be asynchronous to the disk I/O, because data is written to the memory buffer first and transferred to disk later in time. The I/O handling within a compute node is similar. Therefore it can be modeled also with this abstraction. The Unix kernel will process the data, which is handed over by system calls from the application. The data itself is stored in application memory or in system memory (cf. Figure 2.2). GPFS starts an own client daemon and allocates a page pool in memory to maintain file system operations. The page pool is used for managing file-system pages, which are read or modified by client processes. In contrast, Lustre uses the system I/O memory caches to store data, which have to be transferred between compute nodes and file system.

Further, the model can be applied also to the file system. A file system consists typically of multiple server nodes, which are connected to the file system client daemons on compute- or I/O-nodes. The server nodes manage also the file system disk storage, which is attached to these nodes. In principle, the server has the same functionality as I/O nodes. The data will be transferred between disks and connected client nodes, using an internal memory buffer.

The data transfer in parallel I/O can be modeled by a network of these abstract I/O components. This network will contain compute nodes, I/O forwarding nodes, and file-system nodes as depicted in Figure 2.2. Typically, the number of components will decrease on the way from application to file-system disk. Multiple incoming I/O streams will be serialized to a smaller number of outgoing I/O streams. Those points in the network are therefore typical bottlenecks at large scale.

Metadata plays a special role in this model: Depending on the underlying file system, metadata will be handled either together with the file data or it will be handled on separate servers. In the first case, the network can also reflect these operations. In the second case, additional

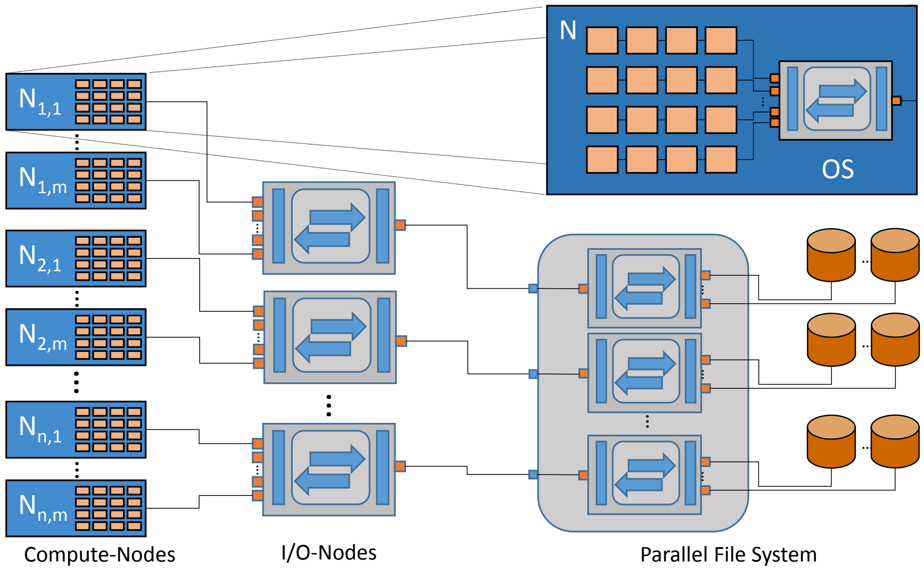


Figure 2.2: Abstract model of I/O network, compute node, and file system.

servers and meta-data storage have to be added to the network as new abstract I/O nodes. For example, GPFS has its own meta-data server with own disk storage, but meta-data will be handled directly by the client daemons itself (cf. Figure 2.3).

2.2 Scalability of Parallel Task-Local I/O

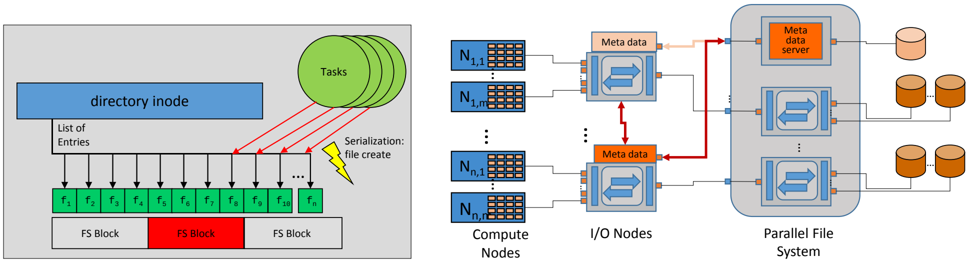
The main reason to use task-local I/O in parallel applications is that the I/O can be easily implemented by using built-in function calls (e.g., with write/read-calls of POSIX, ANSI C, or Fortran). No additional I/O libraries are needed. These I/O calls are serial and they are not aware of the collective nature of application I/O. This leads to independent I/O streams, which are characteristic for task-local I/O in that way that the I/O calls are not synchronized between the tasks of a parallel application. Each I/O stream has one source (e.g., a process) and a destination, which is the individual file on the file system. Therefore, in terms of parallelism, task-local I/O can be seen as *embarrassingly parallel*.

On an ideal I/O infrastructure and file system, this approach should give the best performance, since waiting time at synchronization points between I/O tasks will not occur. Depending on the underlying I/O infrastructure, this type of I/O will give the best I/O throughput on a system. However, the I/O throughput can be limited if the I/O infrastructure has nodes/hubs on which multiple I/O streams are interleaved.

Another downside of the traditional parallel task-local I/O approach is the large number of files. They can cause problems on the file system and are difficult to handle by users. The next two sections will cover these aspects.

2.2.1 Parallel file creation

Trying to create tens of thousands of files simultaneously in the same directory may be serialized on the metadata servers. For example, on the IBM Blue Gene/Q system JUQUEEN described later in this document, the parallel creation of 1.8 million files in the same directory will take about 13 minutes. In general, parallel file systems handle metadata in a different way than the file data itself. Two types of metadata have to be considered for file creation: the inode of the created file and the inode of the directory that contains the files. The first one is not critical, because, the file inode is only accessed by the task that creates the file. In contrast, the inode of the directory is shared by all tasks. The directory's inode stores the pointers to the inodes of files in this directory in a list data structure (cf. Figure 2.3a). Typically, the inodes are stored on special metadata disks, and can therefore be seen as special file objects. The file system has to serialize the access to these objects to ensure consistency (e.g. by file locking). Depending on the file system, the management of the directory's inode is done by different components. Lustre has a special metadata server (MDS), which manages all inodes. GPFS has implemented the delegation of the responsibility for the management of a file to the first client that accesses an inode. As shown in Figure 2.3b, one of the GPFS clients has control over the inode; all other clients have to contact this client to get access to the inode. In addition, GPFS implements load balancing for directory inodes by distributing the inode over



(a) Inode update (b) Inode control delegation in GPFS

Figure 2.3: Schematic view of parallel file creation. Tasks creating a file have to add an entry to the inode of the directory. As the directory inode exists only once, concurrent file creation of multiple tasks has to be serialized. GPFS optimize metadata handling by delegating the control over metadata to the first GPFS clients that opens a file.

multiple file-system blocks and distributing file entries across file-system blocks by computing hash values from the file name. This optimization allows moderate parallel access to the inode, because the file-system blocks will be locked separately (GPFS FGDL, Fine Granular Directory Locking).

Figure 2.4 shows the timings for parallel file creation on JUQUEEN in a directory on the GPFS scratch file system (blue curve). The time for file creation scales approximately linearly with the number tasks. The file creation rate (red curve) increases first from 1400 files/s to a maximum 3800 files/s at 128k task. The rate decreases again to a value around 2200 files/s at larger scale, which seems to remain constant up to the full system. A reason for the decrease of the file creation rate can be that the parallelization strategy of GPFS FGDL scales only up to certain number of tasks, which seems to be reached on the JUQUEEN file system at 128k tasks.

The measurement results for file creation show impressively that task-local file I/O is not

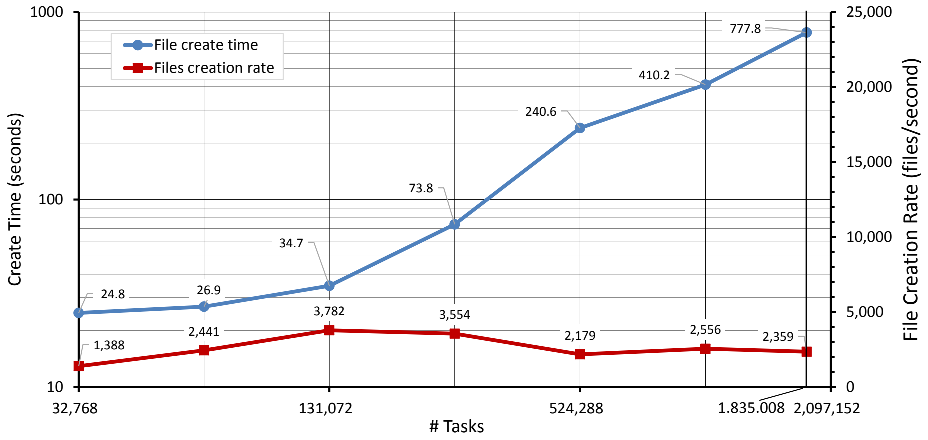


Figure 2.4: Parallel file creation in one directory of the GPFS scratch file system on the Blue Gene/Q system JUQUEEN.

applicable for massively parallel applications. When using this I/O pattern for output files, the creation time has to be spent for each new set of files. Although checkpoint files can be reused after creating it at program start, in general, massively parallel applications have to abandon the traditional task-local file approach and need to implement parallel I/O to shared files.

A popular workaround for the parallel file creation issue is to pre-create a reasonable number of sub-directories, and to distribute the files over those sub-directories. The inodes of the sub-directories are then managed from multiple GPFS clients in parallel. In this case, the number of directories should have the same order of magnitude as the number of tasks, to guarantee low file creation overhead. However, at large scale, this technique only multiplies the manageability issues of using task-local files as it increases the number of inodes needed for directories and files. In addition, this workaround only shifts the problem to the parallel create-operation of the sub-directories in the same parent directory. 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 number, names, and locations of the files, needing some form of agreement between the application and the script. Furthermore, the large number of files and directories has to be handled in all following post-processing steps of the simulation. These issues are described in the next section.

2.2.2 File management

Even with those workarounds described in the last sections, large numbers of files severely complicate file management on different levels. 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 post-processing step, for example, using the `tar` command, also has disadvantages both in terms of the time needed to perform the operation and the at least temporary duplication of the required storage space. Not only access from a parallel application to individual files, but also from serial tools will become a bottleneck at large scale, as typical Unix-commands like `ls` or `tar` will process the files in a list-based execution order. The large execution time, which is caused by metadata operations (e.g., file open), will grow linearly with the number of files. This makes the administration of directories with tens of thousands of entries without support for group operations and automated filtering ineffective. Besides the high complexity of managing large numbers of files, large-scale file operations can cause side effects including temporary service disruptions that are noticeable by other 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 summary, the file management issues described in this section reveal another important reason to abandon task-local I/O patterns. As a solution, these task-local I/O patterns have to be replaced with shared-file I/O patterns.

2.3 Scalability of Parallel Loading

As described in the introductory section 1.4, the demand of parallel applications for dynamic linking and loading becomes more popular with increasing software complexity and the adoption of new software concept like Python-script driven applications on parallel HPC systems. Despite the described benefits, there are serious scalability problems with most modern loading mechanisms. Dynamic linking defers the process of locating object code and resolving its symbols to runtime. To load a library, most loaders first search in a list of file-system locations until the target library is found, and then they load the library into memory. This mechanism does not differ significantly from the method used by the MULTICS [22] operating system that introduced dynamic linking in 1964.

While this load method is suitable for a single-node machine, it breaks down in parallel applications at larger scale. Applications may have over a 100,000 concurrent processes. Typically, processes load the same libraries they depend on simultaneously, but not synchronously. In addition, modern supercomputers typically lack node-local storage, and even large parallel file systems cannot quickly service millions or billions of small, simultaneous I/O requests. As we will see later in this section, loading an application initiates an I/O storm that may manifests like a denial-of-service attack. Whereby the file system becomes unavailable and dependent jobs idle until all requests are cleared. In the following sections, we will analyze that issue in more detail and show examples for dynamic loading on large-scale systems like LLNL's Sierra cluster or the Blue Gene/Q system JUQUEEN at JSC.

2.3.1 Scaling issues

As described in the introduction, the GNU loader is part of process startup. Therefore, loading is performed individually on each process of parallel applications. The loader uses the serial POSIX I/O API to interact with the file system. That interaction consists of two types of operations: Searching and loading the library files.

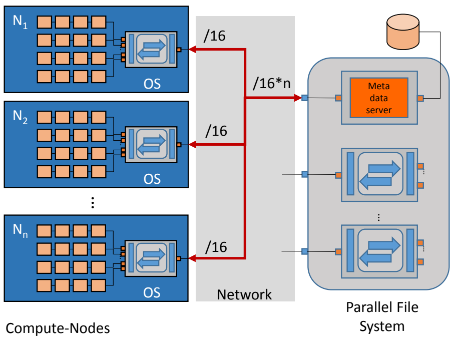
Library searching

The dynamic loader looks up libraries by searching a list of file-system locations. This list of directories consists of system defaults and user-level environment settings (library search paths). The latter one allows users to decide at runtime about libraries (late binding). In this way, it supports a flexible user environment. By simply modifying the search path, users can run the same executable with an updated or improved version of a library without recompiling the application. The concept has remained largely unchanged since MULTICS. However, this algorithm requires a large number of file-system operations, which are not a problem for a sequential program running on a local file system. Typical runtime environments provide a default list of search locations that consists of more than ten directories and each can contain a large number of library files.

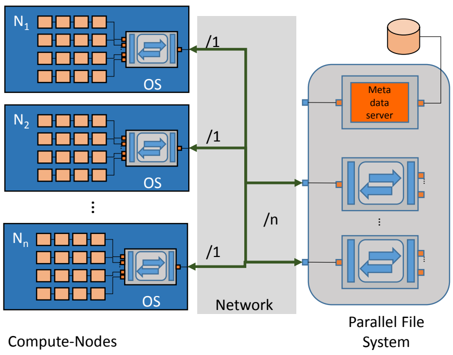
A parallel application with P processes, L library dependencies, and D directories in its search path will perform

$$n_{\text{search}} = O(PDL)$$

file-system operations simply to locate shared libraries. The GNU loader uses the POSIX `open` function to test the existence of files. On Linux clusters, this function will directly interact with the file system. Hence, it will potentially cause a file-system storm at large-scale. Figure 2.5a illustrates this for typical Linux clusters running Lustre or NFS based file systems: Each task of a SMP node will issue the open call. The buffering mechanism of system software is able to cache file data in local memory, but that is not the case for metadata. Typically, local memory buffers do not store information about existence of a file in a directory (*positive/negative look-up*). Therefore, the open request will be forwarded to the files system. In case of a Lustre file system, all requests will arrive at the metadata server (MDS), which has to handle them. Then the MDS has to send back the same number of responses to the clients.



(a) Look-up library



(b) Load library

Figure 2.5: Schematic view of look-up and load operations for an library on a Linux cluster with n nodes using a Lustre file system. The number of requests i per library file are shown for each path ('/i').

Library loading

Libraries are loaded by the GNU dynamic loader of each process individually. Consequently, for parallel application with P processes and L library dependencies this will result in

$$n_{\text{load}} = O(PL)$$

load operations during application start-up. Figure 2.5b depicts how these load requests are handled on Linux clusters. The Lustre file system is able to use the page cache in local memory to buffer data from the file system. Therefore, an SMP-node will issue only one load request if cache can be exploited. Overall, n load requests from n compute nodes will arrive on each file-system server (OST) which holds data of this corresponding library file.

Caching on the local node is done through the operating-system page cache. The GNU loader will first open the library file with the POSIX open call and afterwards map the contents of the file into the process address space with the `mmap` function call. Memory access within the address space of the library will cause only a load operation of the corresponding memory page if it is not already available in the page cache. To be more precise, memory pages in library sections, which are used in read-write mode (e.g., data sections), have to be duplicated, whereas pages in read-only sections can be referenced directly. This requires that the operating system supports shared page access among the processes of a node.

With this optimization and the use of N nodes (P/N processes per node), the number of load operation can be reduced to

$$n_{\text{load,cache}} = O(NL)$$

operations. This improvement will not change the complexity of the application loading, because the number of tasks per node is constant and small (e.g. <64) compared to the number of nodes.

Nearly all major runtime environments in use today utilize these algorithms to locate and to load executable code. Languages like Python and Java use search paths to find and to load modules. Major OSs such as Mac OS X and Windows also use search paths in their dynamic loaders. When run in parallel, these approaches produce $O(PDL)$ search operations and $O(NL)$ load operations, which produce a high load on the file systems. The following section will demonstrate the resulting scaling issue with experimental results.

2.3.2 Dynamic loading on a Linux cluster

To demonstrate the scaling issue of dynamic loading at large-scale, a parallel benchmark was run on two different Linux clusters: Sierra at LLNL and JUROPA at JSC. The first cluster Sierra is equipped with 1,856 compute nodes, each with 12 cores. Data is stored on an NFS or on a Lustre file system. The test environment on the cluster will be illustrated in more detail in Section 6.4. Figure 2.6 shows the results of scaling Pynamic, a benchmark that stresses the dynamic loader (cf. Section 6.2). The benchmark loads in this test 495 libraries with a total size of 1.1 GiB. In the first test, the benchmark loads the libraries from NFS. The overall runtime increased rapidly, and this measurement had to be stopped at a small scale to prevent an I/O storm that would affect other users on the system. This demonstrate the poor parallel support of NFS. It makes dynamic loading impossible when using more than 5 % of

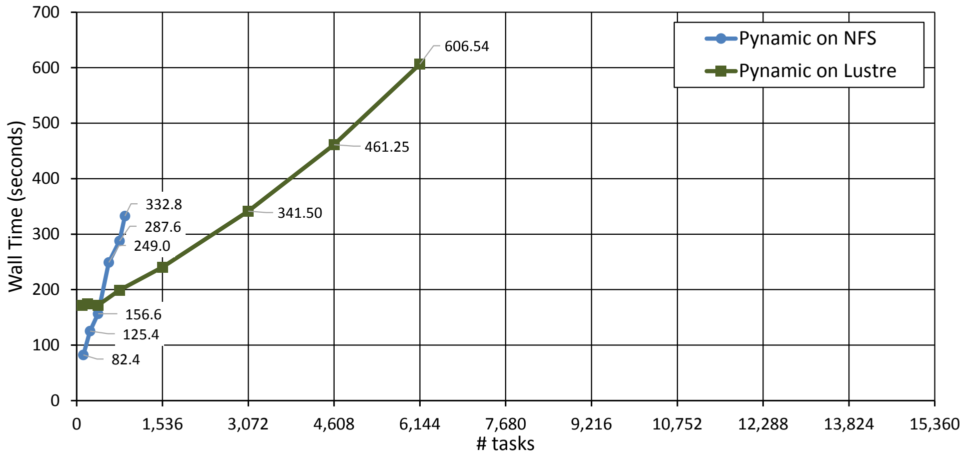


Figure 2.6: Results of Pynamic benchmark runs on Sierra (12 tasks per node, 495 shared objects, 215 utility libraries, 1.1 GiB library data).

the system. This is noteworthy, because system files, including system libraries, are often stored on a central NFS file system. The benchmarks on Sierra ran during production time concurrently with other applications. Therefore, the tests were repeated multiple times and the local memory caches for the file systems were reset before program execution. This gave more reproducible results.

The second test was run with libraries stored in the Lustre file system. The results show a more linear growth in the runtime. The tests were performed with up to 512 nodes (6,144 processes). In general, the parallel file system is better suited for this kind of file operation. However, parallel HPC file systems are typically optimized for heavy write operations with large files and data parallelism. Parallel read operations to small files and a high metadata rate are not the typical file access pattern for which the file system is optimized. The measurements show that using the Lustre file system can be a partial solution to the loading problem, but only up to a moderate number of processes. The tests had to be stopped at 30 % of the system size because the performance of the Lustre file system was already starting to degrade at this scale. The improved scaling of the benchmark can be explained by the higher read bandwidth of library data and the use of local memory by Lustre to cache and share the data. Similar to NFS, the look-up requests are all sent to one metadata server (MDS) and are potentially a reason for the exponential growth of the load time with an increasing number of tasks. To verify that metadata operations occur in a high count during runtime, the benchmark program was instrumented to count the number of corresponding system calls. For the above benchmark run each task will issue about 5,671 `open` and `stat` system calls. Extrapolating this to the total system size of Sierra (23,328 tasks), the number would increase to more than 130 million system calls. Without optimization of metadata handling, such a high number of file-system operations would overload the metadata server.

The results of the benchmark runs on the second Linux cluster JUROPA are shown in Figure 2.7. Meanwhile, the JUROPA system is replaced by its successor JURECA [54]. The JUROPA system was equipped with 3288 compute nodes, each with 8 cores and 24 GiB memory [55]. In contrast to the Sierra cluster, all file systems of JUROPA were Lustre file systems, including the home file system. All measurements were done within the production environ-

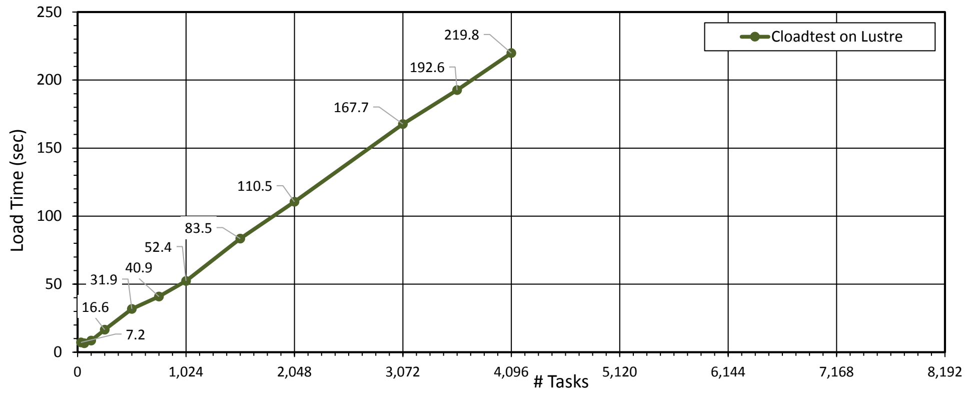


Figure 2.7: Results of the CLoadtest benchmark run on JUROPA (8 tasks per node, 710 libraries, 32 MiB library data).

ment of JUROPA. That implies that other applications were running parallel to the benchmark. Measurements on shared resources like the parallel file system could be influenced by these applications. Contrary to the first tests on the Sierra cluster, it was not possible to reset the local memory caches of Lustre on the compute nodes. Therefore, in multiple similar tests on the same nodes, subsequent runs will benefit from the local memory caches and will need less time to load the libraries. The measurements were repeated at least three times and the best value was reported. The caching on local nodes influences only the timings for data loading, not the look-up timings, because the Lustre file system does no local caching of metadata. All metadata requests are sent directly to the metadata server. On JUROPA, a self-written benchmark program (CLoadtest) was used for the measurement. The benchmark consists of a set of dynamic libraries containing compiled C-code and a main program written in C that loads the libraries at startup (cf. Section 6.1). The implementation of the main program is different to Pynamic, which uses a Python script as a main driver program. Furthermore, dynamic libraries are embedded in Python modules, which have to be loaded additionally.

As data caching on JUROPA could not be influenced, the tests were configured to focus on metadata operations by increasing the number of look-up operations and decreasing the size of libraries. This leads to a benchmark configuration of 710 dynamic libraries with 32 MiB library data per task. Similar to Sierra, the test runs were stopped at a scale of 512 nodes (4096 tasks) to prevent an overload of the MDS. Up to this scale, the load time increases linearly with the number of tasks. This indicates that metadata operations are serialized on the MDS.

Compared to the results on Sierra, the timings show no exponential behavior when scaling to a larger number of tasks. The different configurations of the tests on Sierra (focus on data and metadata) and JUROPA (focus only on metadata) indicate that the exponential contribution might be caused by contention during parallel data loading and not by the look-up operations.

To summarize the results of these experiments, it can be concluded that the startup of dynamically linked applications on Linux clusters does not scale to a large number of processes. Furthermore, it exposes a fundamental scaling problem due to the serial behavior of the dynamic loader. Especially, optimizations of the look-up and the load procedure are required. This optimization can be achieved by caching the metadata in local memory and the exploitation of application parallelism. The next section will show an example of how I/O caching strategies can help improve library look-up and loading on the hierarchical I/O infrastructure of the Blue Gene system.

2.3.3 Dynamic loading on JUQUEEN

The Blue Gene/Q system JUQUEEN has a hierarchical I/O-infrastructure. I/O-requests from the application are delegated to I/O-Nodes (IONs). The configuration of the BG/Q system has an ION-to-compute node rate of 1:128 (production racks). This means that one ION has to serve requests from 8192 MPI-tasks (pure-MPI, 4-way SMT) at most. The I/O nodes are connected to the compute nodes via network links to the internal 5D-torus network and each ION has network links to two different compute nodes, which are denoted as I/O-bridge nodes. I/O traffic between the ION and the other 126 compute nodes is routed over these I/O-bridge

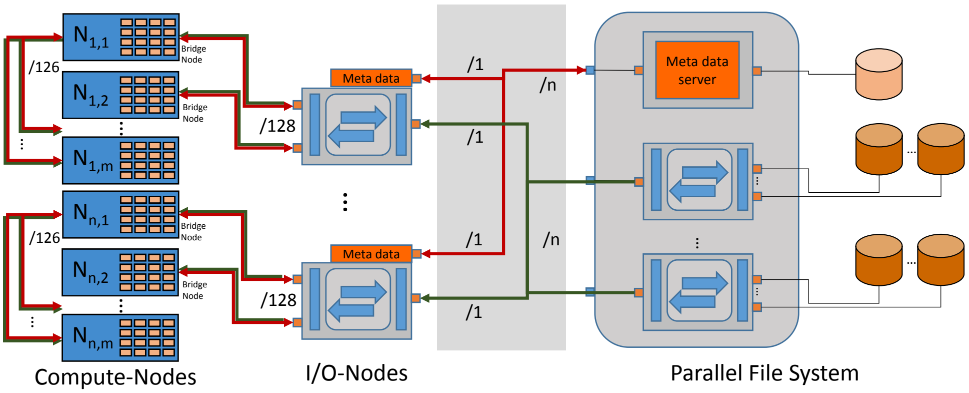


Figure 2.8: Look-up and loading of libraries on a hierarchical I/O infrastructure (JUQUEEN). The number of requests *i* per library file are shown for each path ('/*i*').

nodes on a lower network layer. Similar to application I/O, the dynamic loader uses also the I/O forwarding mechanism to search and load libraries.

Figure 2.8 schematically illustrates the look-up and search mechanism. Look-up requests for files are processed by the GPFS client, which is running on the ION. In GPFS, information about the contents of directories is stored in so-called *directory blocks*, which are handled by GPFS in the same way as file-system blocks containing file data. This means that directory information is stored locally in the GPFS page cache of the ION. The GPFS client will only forward look-up requests for files in a directory when the corresponding directory pages are not in the local cache. Otherwise, positive look-ups (file exists) and negative look-ups (file does not exist) can both directly be answered by the local GPFS client without interacting with file-system nodes. Metadata requests will therefore arrive at the file-system server only once per ION. JUQUEEN consists of 28 racks and each rack is equipped with eight IONs. This results in only 224 requests for a library directory in the case of full system runs.

On JUQUEEN, the same self-written benchmark program CLoadtest for dynamic loading was used as on JUROPA. Figure 2.9 shows the results of runs up to the full system with this benchmark. Again, a benchmark configuration of 710 dynamic libraries was selected, but with a total library size of 400 MiB. This allows the program to run with up to 16 processes on a compute node, which has a physical memory of 16 GiB. The measurements were done for different job sizes, starting with one midplane (512 compute nodes) up to 28 racks (28.672 compute nodes, max. 458.752 cores). The diagram shows a slightly different behavior of load timings compared to the Linux clusters: The timings for a fixed number of tasks per compute node are nearly constant and independent of the total size of the job. The main reason for this nearly constant load time over the whole system is the locality of I/O operations on the I/O nodes. Only requests that occur for the first time on an ION are forwarded to the file system. The number of I/O requests is an order of magnitude smaller than on the Linux clusters. At this scale, the influence of serialization and contention at the file system's metadata server is very small. As already explained, GPFS uses a page pool in the local memory of the ION to cache data-blocks for file and directory data. The size of this cache is about 8 GiB, which is enough to host all library data in our test runs.

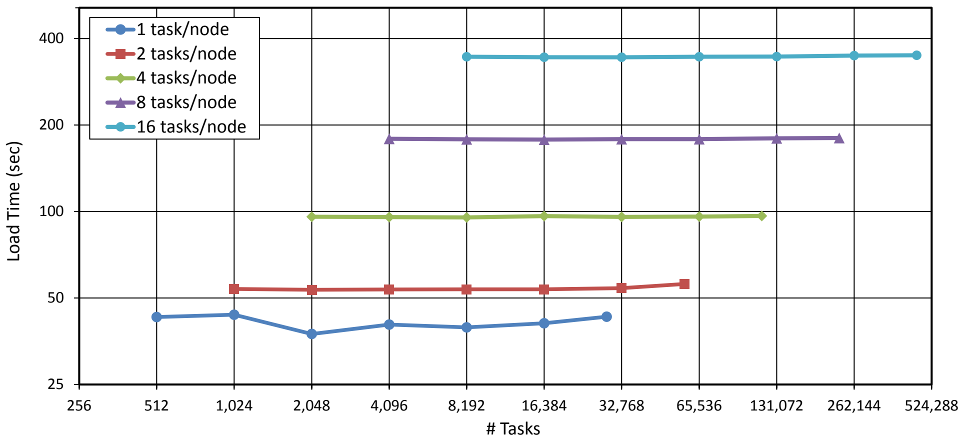


Figure 2.9: Load times of a dynamically linked application on Blue Gene/Q system JUQUEEN for different number of tasks per compute node and scaled up to full system size of 28 racks using the GPFS scratch file system.

The number of tasks per compute node has a significant influence on the load time. The dynamic loader of each process on a compute node will read the library data from the ION, regardless whether other processes have read the data already. In the test case with 16 tasks/node, this leads to nearly 800 GiB of library data, which has to be transferred by each ION to the related compute nodes. With a maximum data transfer rate of about 4 GiB/s, this takes at least 200 seconds. The limited data transfer rate from the IONs to the compute nodes is already saturated with two tasks per compute node. With higher numbers the load time increases linearly with the number of tasks per node.

The redundant read operation on a compute node is unique for dynamic libraries. Statically linked applications or the executable file of a dynamically linked program will be transferred only once to the compute node. The application loader, a component of the Blue Gene runtime system, is aware of application parallelism and uses this information to implement the data transfer in two hierarchical steps. First, the executable data file will be read from the file system and distributed to the compute nodes. Second, the application loader copies the file into local memory in the address spaces of all local processes. These optimizations cannot be applied to the dynamic loader because of its serial nature of library processing. The long load time applies only to those applications that use a larger number of MPI-tasks on a compute node. Through the introduction of hybrid models (MPI+OpenMP) for the parallelization of applications, we see currently less often pure MPI codes that would suffer from the described load delays. Hybrid programs start typically only one or two MPI tasks per compute node, each running a number of OpenMP threads. Load times for such configurations are moderate and comparable to loading of statically linked executables.

As already discussed, GPFS uses a special page pool in memory to cache file system data. However, this cache is not used for system library files such as the libraries for the MPI runtime environment. They are stored in an NFS-mounted file system. The typical size of loaded system libraries in parallel applications is about 50 MiB. At large-scale the total amount of data transfers would jeopardize the NFS server, which we do not see in the measurements. The reason is that also those libraries are cached in the file cache of the operating system on the I/O nodes. With the use of both caches, the GPFS page pool and the NFS system cache, I/O requests caused by system library loading are local to I/O nodes.

On Blue Gene systems without a GPFS file system, users often store library files on an NFS-mounted file system (e.g. home directory). As a comparison to our previous measurements, this situation was emulated on JUQUEEN by selecting an NFS-mounted system directory to store library data, which is hosted by NFS daemons running on the BG/Q service node. As this service node is only used by the Blue Gene system, the results are expected to be better than loading data from a site-wide NFS-server. Figure 2.10 shows the results of running the dynamic loader benchmark in the same configuration as the previous measurements with 710 libraries and a total size 400 MiB. On the NFS directory, the load time is no longer constant. Instead, it increases with a growing number of tasks. The tests are stopped at a size of twelve racks to prevent an overload of the Blue Gene server node. Furthermore, measurements with the same number of compute nodes were performed in one job on the same partition. As a result of this, the load time in the first test (one task/node) is higher than the ones of following tests on same compute nodes. The additional time in the first test is spent to read the library data from the NFS file system, whereas in subsequent runs the file data is already cached in the local memory of the I/O node. The increasing load time in these tests indicates that look-up requests in NFS directories are possible not cached in the I/O node. Moreover, an increasing load on the NFS server was observed during the tests, which is another indicator that look-up requests are not cached on the I/O node. Similar to our experiments on Linux-Clusters with Lustre or NFS, the measurements demonstrate that dynamic loading does not scale on those systems with an NFS file system, although a special hierarchical I/O infrastructure was used.

On the other hand, the hierarchical approach of the I/O infrastructure on Blue Gene presents, together with GPFS caching, a system-level solution for dynamic libraries at large-scale by im-

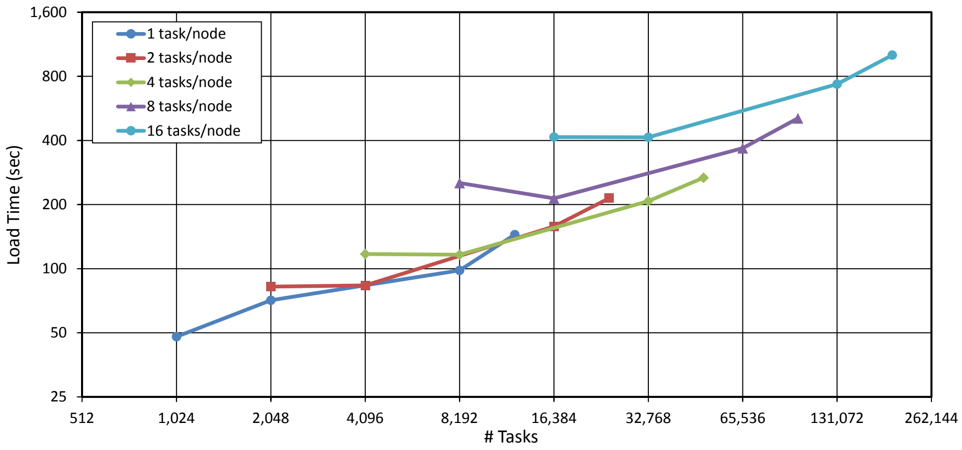


Figure 2.10: Load times of a dynamically linked application on Blue Gene/Q system JUQUEEN for different number of tasks per compute node and scaled up to a size of twelve racks using an NFS file system.

plementing file and directory caching on I/O nodes. Especially the reduction of the number of search and look-up requests issued to the file system is essential for dynamic loading at scale.

This chapter on I/O limitations at large scale demonstrates two different problems, which are caused by the serial design of the underlying I/O functionality: First, parallel task-local I/O suffers from the missing ability of the file system to exploit the knowledge about the parallel nature of applications. Therefore, the file system cannot apply collective I/O processing techniques to those applications. As a consequence, it has to act on a large number of files. On the other hand, the dynamic loader works independently for each process of a parallel application and interacts separately for each library file with the parallel file system. Similar to parallel task-local I/O, the file system is not able to optimize this massive storm of I/O requests with collective aggregation operations.

The next chapters will show with SIONlib and Spindle two solutions on application-near software layers that apply such collective aggregations.

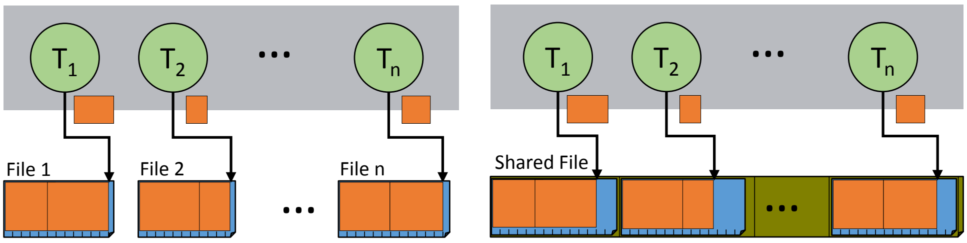
3 SIONlib

This chapter introduces the parallel I/O library SIONlib, which is designed to support parallel task-local I/O at large scale. The general idea behind the SIONlib approach is to use a shared file instead of the individual files that parallel applications create when performing parallel task-local I/O. As explained in the previous chapter, parallel task-local I/O has disadvantages at large scale, which are mainly caused by the management of the large number of individual files. Using a shared file as a container for these individual files avoids this metadata overhead. Similar to the concepts of high-level parallel I/O libraries, the access to a shared file from multiple tasks has to be organized in order to prevent data corruption and bottlenecks through concurrent access.

First, we will discuss how task-local data can be mapped to a shared file and how such a file container can be accessed from multiple tasks. Because of the potentially higher overhead to manage the concurrent access to a shared file, optimization strategies have to be applied to achieve similar I/O bandwidth as traditional parallel task-local I/O. The reasons for the higher overhead and corresponding optimization strategies are discussed in the next section about scalability of shared file I/O. This leads to the definition of the objectives of SIONlib to optimize task-local I/O to shared files, which will be explained in the following sections, including the detailed shared file structure, the design of the SIONlib API, and special features for parallel performance tools that require parallel task-local I/O.

3.1 File Container for Task-Local Data

I/O to shared files is a well-known strategy in high-level parallel I/O libraries. These libraries typically store data in its global view, as described in Section 1.2.2. The libraries require that applications describe the structure and the distribution of data, so that the exact position of data elements in the file can be derived. This generally predefines the file layout, which the libraries have to follow (e.g., MPI I/O). High-level libraries using a self-describing format have more freedom to define the file layout (e.g., *chunking* in HDF5). This is possible, because access to a file is only allowed using the library and the mapping of data to file positions can be derived from the metadata stored in the file. The situation is different for typical task-local I/O patterns as illustrated in Figure 3.1a. These are often implement with POSIX I/O and have byte stream character. The I/O layer is therefore not aware of the data structure and data distribution. Furthermore, as data is seen as a stream of bytes, the size of data elements and their type is also not known. Hence, the desired file format has to support task-local sequential access to the data. In addition, the data has to be byte-addressable in the file to support seek operations.



(a) Task-local I/O (b) Shared file I/O with file container

Figure 3.1: Transition to a shared file format. Traditional task-local I/O manages one individual file per task, whereas a file container reserves one chunk for each task.

Figure 3.1b demonstrates how file data within a file container can be organized. The container reserves a chunk of the file for each task (*chunk-based file format*). At the beginning, the current file pointer of a task will be set to the position before the first byte of the chunk, allowing to perform the original write operations without change. The write operations advance the file pointer within this chunk, which allows writing of data as long as it does not exceed the chunk size. As the chunks are dedicated to a single task, such a file container can guarantee parallel access without conflicts. However, this strategy introduces a new limitation to the application. Whereas task-local files can grow as long as the file system allows this, the above container format limits the size of the chunks. Tasks are not allowed to exceed the initial chunk size. Otherwise, data of the following task will be overwritten, which corrupts the file. Furthermore, each task has to specify the chunk size before the first task writes data to the file. The chunk size has to remain unchanged until the file is closed. These restrictions allow an I/O library to compute the start positions of each chunk after creating and opening the file. It also allows that chunk sizes can vary across tasks.

A further issue of shared file structures is that the metadata information about the chunks has to be stored in addition to the data. Otherwise, an application cannot read the data back again, because it does not know offsets and sizes of the chunks. On a more abstract level, this means that functionality of the file system (inode management of individual files) has to be moved to functionality of the I/O library implementing parallel task-local I/O using shared file containers. Therefore, the file container can also be thought of an *application-level virtual file system*.

An alternative approach to map task-local I/O to a shared file is to interleave data from different tasks (*interleaved file format*). The data blocks of the individual POSIX I/O write operations would naturally determine the granularity of the interleaved segments. The data segments can then be stored in write order, which gives the advantage that the amount of data need not be specified beforehand. Furthermore, the size of task-local data remains unlimited, similar to I/O on individual files. However, as multiple tasks accessing the file in a non-predictable order, the write operations have to be synchronized among the tasks. A typical example for synchronization is to select one task that manages the files, whereas all other tasks have to query file positions for the next write operation from this task. At large scale, such task synchronization will lead to limited scalability. In addition, this fine-grained interleaved storage scheme requires storing additional metadata for each segment (e.g., origin and size). Furthermore, the segments have to be linked with a pointer list to allow seek operations inside the data of a single task. This is similar to the Fortran record format. A third limiting factor for the scalability of this method will be explained later in this chapter. Data to be written from different tasks should be aligned to file-system blocks to prevent file cache invalidation cycles like the issue of false sharing in the memory access of multi-threaded applications [10]. The limitations of this alternative file format let not expect a good scalability at large scale. Therefore, only the chunk-based file format with one chunk per task will be used in the following scalability study and it is selected as the file format of SIONlib.

3.2 Scalability of Shared-File I/O

A shared file as a container for parallel task-local I/O solves the scalability issues of classical task-local I/O, as it reduces the metadata overhead of managing large numbers of files. However, the concurrent access from a large number of tasks to a single shared file introduces new scaling issues. These are caused, for example, by the limited scalability of the file-system mechanism, which coordinates the shared file access among tasks. For this purpose file systems manage *file locks*, which guarantee that only one task at a time can obtain the right to modify the file or a part of it. In addition, the metadata management of a shared file leads to scalability limitations, because the file system manages now only one inode, which stores the file metadata (e.g., the list of file blocks). Changes of this metadata from a large number of tasks introduce a new bottleneck. We will discuss these issues in the following, especially for the chunk-based file container scheme.

3.2.1 File locking (GPFS)

The POSIX standard requires that a file locking mechanism is available, which allows a program to lock the whole file or a part of the file for one task before starting an I/O operation on this file. This mechanism can grant *exclusive locks*, typically needed for write operations, and *shared locks*, which multiple tasks can use for concurrent read operations. Depending on the file system, one or more daemons are responsible for managing the locks. For example, GPFS implements a distributed token management for file and byte-range locking. GPFS clients, requesting a lock for a region in a file, have to request a *token* from the GPFS client that owns a lock for a file region containing the requested region. The responsibility for the requested region is then delegated to the requesting GPFS client. Depending on the implemented strategy for shared file access, the number of requests and delegate operations will grow with the number of tasks accessing the file. The number will increase linearly when applying the chunk-based container scheme, as a task only has to request a lock once for its chunk. The number of lock operations will grow much faster when applying the interleaving strategy, where a task owns a large number of small segments within the file. However, both strategies will lead to a possible bottleneck at large scale.

A strategy to avoid this bottleneck is the parallelization of the lock management, which can be done on user level by partitioning the shared file into multiple physical files. File locking for each physical file is independent from the file locking of the other files and has to scale only to a limited number of tasks accessing this file. Therefore, this strategy is considered for the design of SIONlib and will be described in Section 3.4.

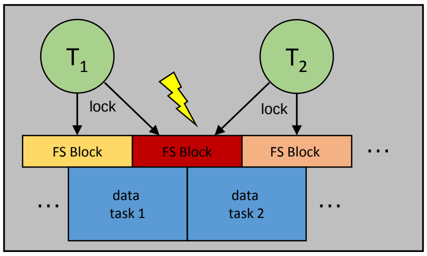
Furthermore, GPFS provides a special feature to push information about I/O access patterns directly to the lock daemons of the file system (`gpfs_fcntl`). This feature can be used for optimizing file locking for the chunk-based file container format as it ensures that information about position and size of the chunks is available at open time. As a result, no further request and delegate operations are needed for file locking [46].

Alignment

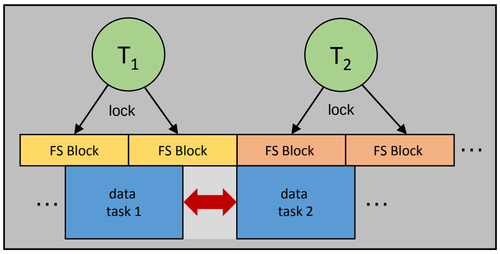
Most of the components of a file system on the server and compute nodes have an internal memory cache to buffer data that has to be written to disk or that was read from disk. In case of write operations, data can reside in this memory buffer as long as the corresponding client has a write lock on the data region in the file. This feature can be used to aggregate the data of small write operations in the memory cache and send it to the file system later as one big block. GPFS, for example, allocates a page cache as a memory buffer on each client, which mirrors file-system blocks in local memory. Because such blocks can only be handled in one piece, a GPFS client has to invalidate a full page in the memory cache when it has to release a write lock on a region that overlaps with the data region of this page. To synchronize page handling with the lock handling, GPFS restricts byte-range locking to the granularity of file-system blocks.

The approach to dedicate a chunk of the file container to one task guarantees that tasks do not concurrently access identical parts of a file. However, 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 these chunks. The situation is similar to the false sharing of cache lines in a multiprocessor. As depicted in Figure 3.2a, the GPFS client of the second task has to wait until the GPFS client of the first task has flushed the file-system block to disk and has released the corresponding write lock. Subsequently, the second GPFS client gets the write lock and can read the file-system block in its page cache to add data to it. Consequently, write accesses to the same file-system block are serialized through this mechanism.

To avoid this limitation, the chunks have to be aligned with file-system block boundaries as shown in Figure 3.2b. This guarantees that only one task accesses a file system block. The file-system block can stay in the page cache as long as a caching algorithm does not purge it out.



(a) No alignment



(b) Alignment to file-system blocks

Figure 3.2: No alignment of data to file-system blocks in a dense shared file leads to a serialization of data access from multiple tasks, whereas the alignment of chunks supports parallel access.

Furthermore, due to these concepts, other tasks will not request write locks for this block and a GPFS client does not have to release an existing write lock. This omits lock communication between the clients and read-modify-write cycles on a client during write time. File-system access to a shared file container with aligned chunks is therefore efficient. To verify this, we will discuss the results of a measurement on JUQUEEN comparing I/O bandwidth of writing and reading data to unaligned and aligned chunks in Section 4.2.3.

The Lustre file system handles the byte-range locking differently. It has no restriction on the granularity of file-system blocks. On the other hand, caching of file data is done in the file-system components with different granularity. The data blocks in the client memory cache have the same size as the memory pages of the operating system, whereas on the server side the file data is partitioned in file-system blocks of the underlying local file systems on the OSTs. Furthermore, file data is distributed over the OSTs in portions of user-defined size (stripe size). Especially with the latter partitioning, Lustre is able to delegate part of the file metadata and lock management to individual OSTs. From this perspective, an alignment of chunks to these blocks should result in good I/O performance.

3.2.2 Number of tasks per files

Running shared file I/O with a very large number of tasks introduces a new bottleneck in the handling of file metadata, which is stored in the file inode. Similar to other file systems, GPFS uses the inode structure and *indirect blocks* to manage the addresses of the file-system blocks of a file. For small files, GPFS stores references to file-system blocks directly in the inode. As files become larger, the space in the inode structure is exhausted and the pointers are stored in additional blocks, the *indirect blocks*. In this case, the inode contains only references to the indirect blocks, as indicated in Figure 3.3. The inode structure exists only once for a shared file. This requires that the access to the inode and its indirect blocks from multiple clients has to be coordinated with write locks.

GPFS optimizes the management of the inode structure by delegating the responsibility to the first client that opens or creates a file. The corresponding GPFS daemon becomes the designated metadata manager of the file and owns the exclusive lock for the inode and the indirect blocks [44]. All other clients that access the file will sent their update requests to this metadata manager (e.g., to add new file-system blocks). As depicted in Figure 3.3, only the metadata

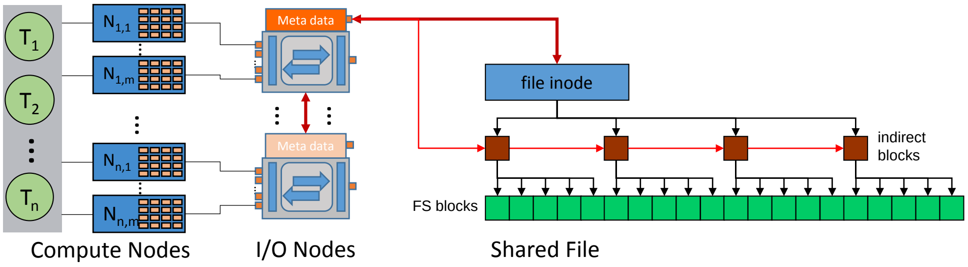


Figure 3.3: GPFS inode handling of a shared file on a hierarchical I/O infrastructure. References to file-system blocks of a shared file are stored in the inode or in indirect blocks. In GPFS, these data structures are managed by the first GPFS client that opens the file.

manager has to communicate with the file system servers; all other clients communicate internally to this client and cause a cross-traffic between the I/O components of a HPC system with their update requests.

In case of multiple (task-local) files, the delegation of metadata responsibility implements a separation and parallelization of metadata handling, because potentially different GPFS clients will handle the metadata of the files. This is a good strategy in the case of a file system with multiple concurrently running applications performing I/O to individual files. However, it does not help for a single application using shared-file I/O at large scale, as only one GPFS client has to manage the update requests of the inode structure for all processes accessing the shared file. Although the GPFS strategy of the distributed inode and lock management is faster than traditional parallel access with explicit lock handling, it will become a bottleneck a larger scale. This will be demonstrated in Section 4.2.4 with a measurement of shared file I/O on JUQUEEN. It shows that the I/O write bandwidth to a single file is degraded using 32k tasks or more.

The aforementioned strategy of GPFS to separate the metadata management for different files is used as a motivation for SIONlib's *multi-file* approach. Instead of writing to one physical file, the logical file container is distributed over multiple physical files (cf. Figure 3.4). With this optimization, the metadata handling will be parallelized over the GPFS clients that manage the inodes of the physical files. This helps especially on a hierarchical I/O infrastructure like the IBM BG/Q configuration to keep the metadata management local to one I/O-node and its assigned application tasks. With such an optimized mapping of tasks to physical files, GPFS clients need not communicate with each other to exchange metadata and no additional cross-traffic between I/O-nodes is produced. This approach reduces the complexity of the metadata management for the shared file container from a global problem size of all application tasks to a local problem size, which is limited by the number of tasks per file or I/O node.

SIONlib's *multi-file* approach with its separation of I/O handling improves the I/O performance also for the Lustre file system. Lustre supports this with the distribution of files over the OSTs. In Lustre, a file consists of a number of file chunks, which are stored on the local file systems of the OSTs according to the selected file striping parameters. As they are normal files, they have their own inodes, which leads to local metadata management. Therefore, a multi-file container should be distributed using one physical file per OST. However, this cannot avoid the complete overhead because file locking and part of the metadata has to be adjusted globally.

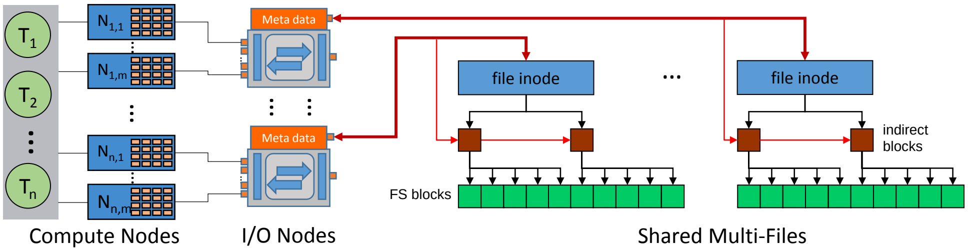


Figure 3.4: GPFS inode handling for a shared multi-file on a hierarchical I/O infrastructure, where each I/O-node manages its own physical file.

The strategy to split a shared file into multiple physical files in the file system requires the virtualization of these files on the software level. The mapping of tasks to chunks of the file container has to be extended to map the task to the corresponding physical file also. This increases the complexity of I/O management in the application or in the I/O-library. More abstract, this reflects the general strategy of the shared file container to move functionality from the file system to a higher software level to improve scalability.

3.2.3 Shared files and data size

In traditional parallel task-local I/O, only one task accesses an individual file. The application typically appends data to the end of the files, so they grow simultaneously on the file system with each write operation. In contrast, a shared file container stores a number of such linearly growing chunks in a large collective file and reserves a pre-defined space in the shared file for each task. The file container is therefore a sparse shared file, where the chunks of tasks can be empty, partly filled, or completely filled (cf. Figure 3.1). The gaps between chunks can grow further if these chunks are aligned to file-system block boundaries, as proposed in the previous section.

The usual file system techniques to reduce wasting of disk space for small files cannot be applied directly to such sparse shared files. For example, GPFS stores the data of very small files directly in the inode structure instead of allocating a separate file-system block for it. The size of files up to which this will be done depends on the file-system configuration and is set for the scratch file system on JUST to 3968 bytes. Furthermore, GPFS manages *sub-blocks*, which have a size of $1/32$ of a file-system block. Files that are smaller than one file-system block are stored in a fragment, which is built of one or more of these sub-blocks [46]. Since file-system blocks on scratch file system are typically large (for example, 4 MiB on JUST), the file system can preserve a large amount of space in the file system with these two techniques. The total size of a sparse shared file typically exceeds these limits. Therefore, GPFS cannot apply these optimization techniques to these files. Consequently, a file-system block is allocated completely on disk, although only a small part of it is used for data storage.

The fragmentation of sparse files on disk can also influence the usage of the memory based disk caches as depicted in Figure 3.5. For example, the page pool of GPFS is working with pages of the same size as file-system blocks. Pages in this cache are directly mirrored from the file-system blocks on disk. Furthermore, GPFS will only transfer complete pages to the

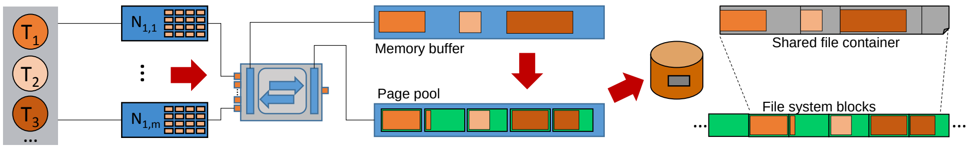


Figure 3.5: GPFS page handling for sparse shared files in a system with a hierarchical I/O infrastructure like JUQUEEN. On I/O nodes, file data is received in an incoming memory buffer and transferred by the operating system and the GPFS client to the pages in the page pool. File-system pages are transferred as full blocks to the file system, disregarding their degree of filling.

file system, when data has to be flushed out of the page pool. In this way, the number of file-system blocks and not the amount of application data that is stored in the file-system blocks, limits the data transfer bandwidth between I/O nodes and file system. Therefore, the efficiency of parallel I/O using a shared file container depends on the efficiency of its internal alignment. Best results can be expected, if the size of task local data is a multiple of the file system block size, which guarantees completely filled blocks. The efficiency will be degraded, if the file contains blocks that are nearly empty or only partly filled. The percentage of these blocks decreases reciprocally with the application data size. This means that applications with large task-local data will not suffer from this fragmentation.

In contrast, applications with small task-local data will suffer from the shared file container approach and therefore need further optimization. The container would be denser if the alignment of task-local data was abandoned. However, as discussed previously, in this case file lock handling would lead to a serialization of I/O operations of different processes when accessing the same file-system block. The second possibility is to reduce the number of tasks writing to the file, which would directly reduce the number of alignment points in the file. Consequently, file fragmentation would be reduced. Again, this optimization requires more functionality on application or I/O-library level, because data first has to be aggregated on a number of collector tasks and secondary has to be written representatively for the other tasks into the file. The number of the collector tasks influences the efficiency and is – together with the number of tasks and the data size – an input factor for the calculation of the trade-off between I/O performance optimization and disk space optimization. This *two-step I/O* method is considered in the SIONlib approach for *coalescing I/O* to optimize parallel task-local I/O with small data chunks at large scale (cf. Section 3.8). The results of comparative measurements of coalescing I/O with different configurations are discussed in Section 4.2.6 of the next chapter *Evaluation of SIONlib*.

Furthermore, the size of the file container can be another issue. When each task is writing a large amount of data, a shared file will have the aggregated size and can exceed the system or site limits for file sizes. In this case, SIONlib's multi-file approach provides a mechanism to limit the size of the individual physical files without limiting the overall size of the virtual shared file.

A restriction of old file-system implementations is that files may not exceed a size of 2 GiB. Data structures in the file system that use 32-bit integer variables to store file offsets or size information (e.g., in the FAT32 file system) cause this technical issue. Although this limitation is not given any more in modern file systems, functions of the POSIX I/O layer uses such 32-bit integer variables per default in parameter data structures (e.g., the `stat` function). Therefore, programs requesting file sizes with more than 2 GiB have to enable special *large file support* during compilation to change the variable type to 64-bit integers in order to be able to handle large file offsets. As large file sizes have to be foreseen, the large file support is incorporated into the SIONlib library layer.

3.2.4 Shared files and threaded applications

Although hybrid applications often have implemented their parallel I/O in the MPI layer, there is a need to support hybrid applications and tools that use I/O operations in OpenMP parallel regions. One example is the performance analysis tool Scalasca (cf. 1.3.3), which supports in its version *1.x* hybrid applications with a fixed number of OpenMP threads and requires that each thread writes and reads trace data itself. With the restriction that the number of threads is known in advance when the file is created, the proposed container format can support parallel task-local I/O from hybrid applications without modification. Instead of assigning a chunk of the file to each MPI task, a chunk is assigned to each OpenMP thread. In this case, all I/O operations have to be performed inside the parallel region, where all OpenMP threads are forked and active.

The restriction to require that the number of threads is known in advance excludes the category of applications that define the number of threads dynamically during runtime. For example, the application can define the number of threads depending on the computational algorithm used, input data requirements or the maximum available threads on the compute node. To support such applications, the current scheme of the shared file container has to be enhanced in a way that chunks can be created and added to the existing container on the fly after opening the file. Such changes to the file container structure would require that updates are communicated directly to all other threads in the application, which introduces a potentially large number of synchronization points during runtime. Similar to the implementation with interleaving records, the overhead to guarantee a consistent view of the container would limit scalability and will therefore not be considered for SIONlib.

Partial support for multi-level parallelization with a variable number of threads can be implemented if parallel write operations can be performed outside the parallel region. The organization of the file container could then be implemented in following way: chunks will only be assigned to MPI tasks. Similar to the coalescing approach, only one thread per MPI task will write the data on behalf of all threads. To allow a separation of the different data streams at read time, the data blocks have to be annotated with their corresponding thread number. This strategy will fix the number of chunks in the file container to the number of MPI tasks on the outer parallelization level. Furthermore, parallel read operations can be threaded on the inner parallelization level, because they do not change the container structure. One application example that benefits from this strategy, is the Score-P instrumentation and measurement infrastructure, because it supports hybrid programs with variable number of threads (cf. Section 1.3.3). Score-P realizes writing of trace data on the outer level, whereas reading of trace data is multi-threaded.

3.3 Objectives and Strategy

The major objective of SIONlib is to provide efficient support for task-local I/O patterns at large scale. Recapitulating the findings from the previous discussion about traditional parallel task-local I/O and shared file I/O, SIONlib should fulfill the following goals:

- The general structure of application I/O patterns should be unchanged. Especially, the task-local representation of application data has to be kept. This helps to ease the transition from standard POSIX I/O to parallel I/O with support of an I/O library.

- The library has to mitigate the limitations of parallel task-local I/O on current file systems, which are mainly caused by the large number of individual files. A transition from a file-per-task scheme to a shared file container with parallel I/O is mandatory.
- The solution should also eliminate the limitations of parallel I/O to a shared file at large scale, as described in the previous section. This means that the metadata overhead of accessing one big shared file from a large number of tasks has to be reduced.
- Existing software layers on current HPC systems should not be modified. Furthermore, all components should run in user space. Therefore, it is advisable to use POSIX I/O in the low-level interface to access the file system. On the application level, the solution should be integrated as a library into the parallel application, where it can use the communication layer of the application to exchange data internally.
- Access to the file data should be possible from parallel applications and tools as well as from serial applications. This allows an easy integration into existing workflows.
- The solution should support applications with different requirements with respect to data size and distribution. Examples are the support of small data chunks per task and the support of hybrid applications.

As required, SIONlib has to avoid large numbers of files due to limited metadata scalability. This leads to the basic strategy of SIONlib to use a file container instead of individual files as illustrated in Figure 3.6. Located as an additional software layer between a parallel application and the underlying parallel file system, SIONlib maps a large collection of logical task-local files onto a number of SIONlib shared files. The limitations of shared-file I/O at large scale motivated the design of the multi-file approach of SIONlib, which uses multiple physical files to represent the virtual shared file container. This strategy is described in more detail in Section 3.4, whereas the organization of the file container into multiple files and chunks for each

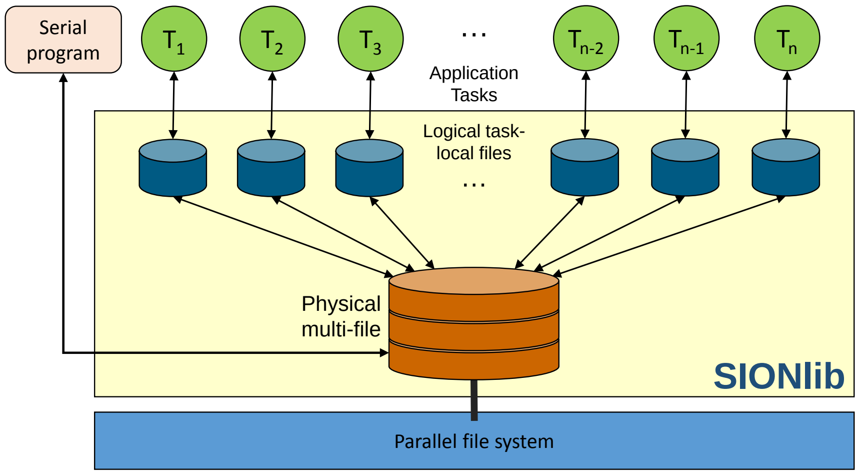


Figure 3.6: File-container concept of SIONlib. A large number of logical task-local files is mapped onto a single physical file (or a small set of physical files), called a multi-file. The multi-file can be accessed from both a parallel and a serial application.

task is described in Section 3.5. To meet the above objectives, applications have to fulfill some requirements, which are described in Section 3.6. For example, SIONlib requires the specification of chunk sizes at file creation time. The library layers and interfaces, which are described in Section 3.7, support the integration of SIONlib into serial programs as well as into parallel applications with different parallelization strategies (e.g., MPI and OpenMP). Moreover, with a special generic interface, SIONlib can be adapted to new parallelization schemes without modification of the library itself. The modular concept of SIONlib and its implementation as a user-space library does not require changes in the software stack of an HPC system. Therefore, SIONlib can be seen as a very simple application-level file system with an API and command-line utilities to access task-local logical files in a shared file container. Section 3.8 describes the design of a special SIONlib feature to support I/O of very small data chunks (*coalescing I/O*). Section 3.9 introduces the key-value containers needed by hybrid applications with variable numbers of threads. Section 3.10 describes additional functionality, which helps to integrate SIONlib into other parallel tools like Scalasca, Score-P, or Vampir. The chapter concludes with an overview of related tools and libraries for parallel I/O addressing task-local I/O.

3.4 Separation of I/O Streams

As described in Section 3.2.2, the number of tasks that write to a shared file concurrently influences the efficiency of parallel I/O. A large number of tasks introduces bottlenecks in file metadata handling. Therefore, SIONlib follows the multi-file approach to separate the data chunks of subgroups of tasks into different physical files to parallelize and optimize metadata handling. Each of these files is an independent POSIX file, whereas all together represent the SIONlib shared file. In this strategy, metadata operations are concurrent on the local level for one physical file and not any longer on the global level.

According to the requirements, the handling of multiple physical files has to be transparent to the user and should be implemented in user space. This implies that the mapping of tasks to physical files has to be defined on the library level and need to be stored persistently in the metadata of the SIONlib container. However, applications should be able to influence this mapping, for example, if application tasks are already divided into subgroups of tasks. In this case, data of different subgroups would be stored in different physical files.

The technical requirement to reduce the number of tasks per file leads to another design approach. The separation of tasks into multi-files can be used to define sets of I/O streams that are local to components of the I/O infrastructure (cf. Figure 3.7). For example, on systems with a hierarchical I/O infrastructure, all tasks that are assigned to one I/O node can be mapped to the same physical file. In this case, only one I/O node manages and accesses the file. Data of a physical file therefore only exist in local memory caches of the corresponding node and cross-traffic to other I/O nodes to adjust metadata is not necessary. Although the global parallel file system will be an end-point of all I/O stream sets, the separation of I/O operations can also be conducted on the file system level. As GPFS delegates the metadata management to the first client that opens a shared file, the metadata management will be performed individually for each physical file by another I/O node. The HPC runtime system typically knows the mapping of tasks to I/O nodes, which allows SIONlib to exploit this mapping for the separation of multi-files. Furthermore, Lustre provides file stripping over the OSTs, where file

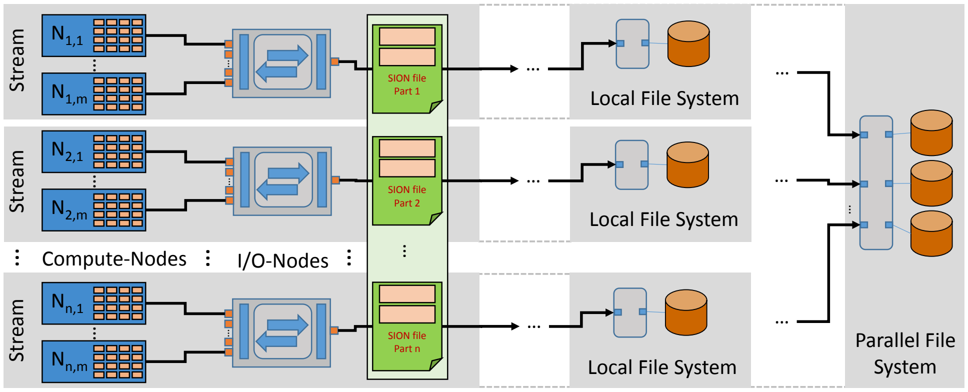


Figure 3.7: SIONlib's multi-file approach supports the separation of I/O streams according to the I/O infrastructure (e.g. I/O nodes) and data locality (e.g. local or global file system).

metadata is partially managed locally on the OSTs. Therefore, multi-files should be separated by assigning the physical files to different subsets of OSTs. In this case, two conditions, the I/O infrastructure and the file-system layout, define the mapping. Fortunately, the application or the I/O-library can define the file-to-OST mapping, since Lustre provides user-level configuration of file striping. Moreover, also the information about the network structure can be used to optimize the file mapping. On inhomogeneous interconnects, multi-files can be defined by grouping those nodes that have a stronger connection and separate those nodes that are loosely connected. For example, some HPC systems have nodes combined in islands, which have a full fat tree interconnect, whereas the islands themselves are connected with less bandwidth between them. The resulting file mapping should assign nodes in the same island to the same physical file.

In the multi-file approach, not only the I/O streams are separated. Moreover, the data is also localized, because a global file system with a global name space is no longer necessary. As illustrated in Figure 3.7, the physical file can reside on local storage, which is only accessible from those tasks that access it. This is possible because metadata management for the virtual SIONlib file is done within the library using application level communication. More research on this locality property of SIONlib's file container is performed within the EIC cooperation with IBM about Blue Gene Active Storage (BGAS, [18]) and in the DEEP-ER project [88]. In both projects, SIONlib will be adapted to use the local storage on the I/O-nodes (BGAS) or on the so-called *cache domains* (DEEP-ER) to exploit locality. Since multi-files are implemented in SIONlib transparently, data on local storage can be migrated to global storage without losing data accessibility. This supports the management of multi-level checkpoints [73], which could reside on different storage levels and be migrated between them.

3.5 File Organization

This section introduces the organization of the SIONlib file container step by step, starting with a simple and preliminary layout, which will be refined as new features are discussed. The file container has a substructure based on chunks, which are assigned to the application tasks. Additional chunks are used to store metadata, which describes the structure of the file container. Figure 3.8 depicts this format of the file container, which starts with a metadata block. The data chunks of the application tasks are placed according to their rank order. The size of the chunks has to be known in advance before creating the file container. Therefore, the application tasks have to specify the maximum requested size as parameter of the file open statement. SIONlib takes this information and reserves the corresponding space in the file. In addition, SIONlib extends each chunk so that its end position is aligned with the boundary of a file-system block. As discussed in Section 3.2.1, this prevents possible congestion when accessing chunks from different tasks.

The metadata block that precedes the data chunks is needed to ensure that the content of the file container can be read again by an application. It stores scalar attributes and arrays with one element per task. For example, scalar attributes are used to store the number of tasks that have written data to the file. Array data is needed to describe the size and the fill rate of chunks. The number of elements of these arrays is known in advance, because the number of tasks writing to the file is required to be fixed after opening the file (cf. next Section). Therefore, the size of the metadata block is known in advance and it can be placed at the beginning of the file. The space for this block is also extended to be aligned with the boundaries of file-system blocks. The metadata block exists only once in the file, and because of efficiency multiple tasks should not access the metadata block concurrently. To ensure this, SIONlib aggregates the metadata on one tasks in its collective file open operation using the application communication layer. In detail, all tasks send their requested chunk size to the master task, which is responsible for writing the metadata block, to calculate the individual start addresses of each chunk, and to return the start position of the chunks to the tasks. With this, the tasks can advance the file pointer to the beginning of the reserved chunk. Because the file creation and the calculation of chunk positions are done during the open operation, no further collective operation is needed until the file is closed. Application tasks can act individually on the file container for writing and reading data. During the close operation, the master task collects the number of bytes written from each task and stores it in the metadata block. The close operation is again collective to avoid the inefficiency of having all tasks writing to the metadata block concurrently.

The presented file format requires that chunks sizes are defined when the file is opened. However, the need to know the total amount of data each task writes is too restrictive for applications that cannot compute the data size in advance. With an extension of the file format, this restriction can be relaxed to the requirement to know the maximum amount of data written in one piece by each task. The file format extension leads to the layout depicted in Figure 3.9. The file container is now organized in *blocks* with each block containing one chunk per task. If a task wants to write more bytes than there are left in the current chunk, it can request a

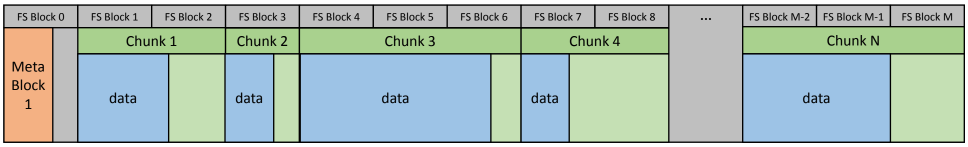


Figure 3.8: Simple and preliminary structure of the SIONlib file container. The chunks are assigned to the application tasks and their size is extended to be aligned to file-system blocks. The metadata block stores information about chunk sizes and fill rates.

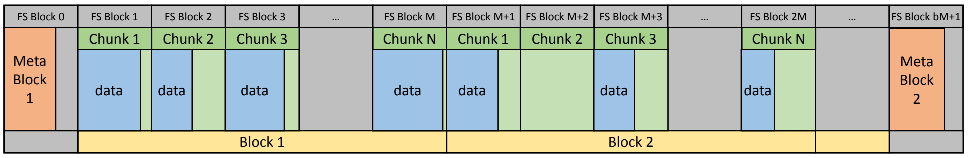


Figure 3.9: Extended structure of the SIONlib file container. The chunks are now organized in blocks, which can be repeated multiple times in the file. A second meta block at the end of the file is needed to store metadata with variable size.

new chunk of the same size. As chunks have a predefined size, the size of a block is also known beforehand and each task can compute the positions of subsequent chunks on its own without the need to communicate with other tasks. It is noteworthy to mention, that this may create substantial gaps in the file container if only a subset of the tasks ask for additional chunks. However, since file systems typically do not allocate space for empty file-system blocks, these blocks exist only on the logical level and not on disk. To avoid their physical materialization, for example, when the file container is copied, the file can be defragmented in a post-processing step with tools provided by SIONlib.

SIONlib needs to store metadata indicating the space used in each chunk without knowing the total number of blocks in advance, the first fixed-sized metadata block cannot be used for this purpose. Instead, SIONlib allocates a second metadata block at the end of the file in the collective file close operation. In this block, SIONlib stores the number of chunks per task and the space occupied by data in each of the chunks. However, appending data to a SIONlib container beyond the initially allocated space after it has been closed would require updating and re-writing the second metadata block. Although this feature is not required for SIONlib, adding it would not pose a fundamental design problem.

As discussed in Section 3.4, SIONlib shared file containers should be dividable into multiple physical files to improve scalability and to use the hardware or software parallelism that is available between the application and the disks. Therefore, the SIONlib file format is extended further to offer the option of distributing the chunks across a user-defined number of physical files (cf. Figure 3.10). Each task is still mapped onto a single physical file, but two tasks may now end up being mapped onto different physical files. The first physical file stores additional data that is needed to manage multi-files (e.g., the number of multi-files) in the first metadata block. In addition, a mapping table is added to the second metadata block of the first file. This table contains the file number of the physical file and the local rank number in this file for each task. Each file is complete in terms of the SIONlib file format. A multi-file contains the two metadata blocks and stores a subset of chunks. This allows SIONlib to dump metadata or to read file data independently for each of the multi-files. To implement this transparency, the first metadata block of each file will maintain additionally a list of global rank numbers, indicating the application tasks that have written data to chunks of this file.

The use of multi-files with SIONlib is optional. Applications are able to use multi-file container or single-file containers without modifying their code. This means that the SIONlib file is identified by the name given in the file open operation in both cases. Therefore, the first physical file is stored under the originally specified name, whereas a consecutive number is appended to the filenames of the other physical files.

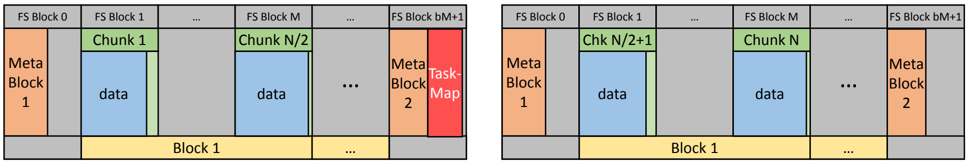


Figure 3.10: Multi-file structure of the SIONlib file container (example with two physical files). Each file is a consistent SIONlib file and stores a subset of chunks. The first file has a third metadata block now, containing a task-to-file mapping table.

Data and metadata is written at different positions in the file and at different times. In cases of failure, this can lead to corrupted data, because data can only be accessed from the file container if the corresponding metadata is available. For example, with a missing second metadata block, the reading application cannot determine how much data has been written by each task, because the number of bytes written is not known and an end-of-file cannot be detected by the corresponding POSIX-call (`feof`), since the end position of a chunk is not located at the end of the physical file anymore. Therefore, file consistency is only given after closing the SIONlib file correctly. Although this limits the usability of data in failure situations, it is not a general problem in typical uses cases of parallel task-local I/O. For example, the consistency of checkpointing data is often ensured on the file level. The corresponding file operations will create a new checkpoint file in each checkpoint step and will delete earlier checkpoint files only after the current operations succeeded.

3.6 Application Requirements

The objectives described in Section 3.3 and the design of the SIONlib file format lead to a set of requirements, that applications have to fulfill before using SIONlib. A first requirement is that task-local data is only accessed by one task, ensuring a one-to-one relationship. While simultaneous read operations to same data are possible, writing from different tasks to the same chunk would corrupt SIONlib's metadata, because metadata will be maintained only by the task that has been assigned to the corresponding chunk.

Furthermore, SIONlib requires that all tasks create, open, and close the SIONlib file at the same time. This is needed, because these operations are collective in SIONlib in order to collect and synchronize metadata among all tasks. This requirement is not given for file write and read operations while the file container is open. Application tasks can perform these operations individually and asynchronously.

In addition, each individual task must know in advance the maximum amount of data that may be written in one piece (i.e., in a single write call). 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. With the information about the maximum chunk size, SIONlib can provide continuous space of this size in the file container. In the current version of SIONlib's API, application tasks are able to get direct access to the internal ANSI file pointer, so that they can use standard ANSI C I/O calls to write or read data from the file. As SIONlib has no control over these operations, the library cannot manage the data in another way, for example, by using smaller chunks and spreading the data across multiple

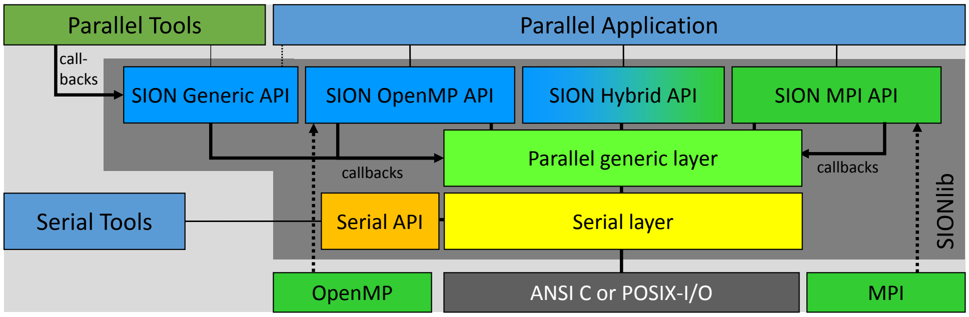


Figure 3.11: Software architecture of SIONlib.

chunks. However, SIONlib also offers its own version of write and read functions for binary data. Although these functions are provided to improve and simplify usability, they can also be used to circumvent this restriction in a more systematic fashion by overlaying the striped chunk structure with a continuous virtual address space. To implement this, SIONlib needs to require to use the own write and read functions. The appropriate API changes are planned for the next major releases of SIONlib.

As already described, all these assumption are realistic for a broad range of applications that use task-local I/O patterns similar to those presented in Section 1.3. For example, code for checkpointing is often concentrated in one routine, which is called at the same time from all tasks. Collective I/O operations and the pre-calculation of data sizes are therefore easy to implement.

3.7 Software Layers and APIs

Although SIONlib is designed as a parallel I/O library to be integrated in parallel applications, it provides additional interfaces for parallel and serial pre- and post-processing tools. To reduce the software complexity and limit the dependencies between the software components, the library is organized in multiple abstraction layers. As depicted in Figure 3.11, on the top layer SIONlib provides the parallel APIs, which support applications that are parallelized with MPI, OpenMP, or, in a hybrid variant with both parallelization paradigms. Besides this, an additional parallel generic API is designed to support the integration of SIONlib into parallel tools like Score-P (cf. Section 3.10).

The parallel functionality of SIONlib is implemented in a generic layer below the application APIs. It includes the management of metadata and the maintenance of the metadata blocks in the SIONlib files. This layer only needs support from the application's communication layer for the aggregation and distribution of information among the tasks. The generic layer does not directly use the required communication functions, which are only available in a higher software layer. Instead, the parallel API layer have to provide these via callback functions. With the use of the callback technique, the core layers of SIONlib can be designed without dependencies to higher software layers or external parallel libraries like MPI [16].

The task-local and serial functionality is encapsulated in the serial layer of SIONlib. Functions of this layer will be called from the parallel generic layer and the serial API. Serial tools are needed for example to integrate SIONlib into existing workflows, because the pre- and post-processing tools in those workflows have to be adapted to write or read the SIONlib file format. Alternatively, conversion tools can be implemented on the basis of SIONlib's serial API, which convert data between the application specific file format and the SIONlib file format. Additionally, SIONlib provides a number of serial commandline tools to manage the file container.

I/O operations on the physical files that represent the SIONlib shared file container, use ANSI C or POSIX I/O calls. The user can select, which of these two should be used. This is optional, because, depending on I/O pattern and available memory, ANSI C I/O operations with their internal buffering mechanism have some advantages: using these I/O calls, which are directly available in C programs, SIONlib does not need to rely on external I/O libraries like MPI I/O. Only the upper API layer has dependencies on MPI and OpenMP, all other layers can directly be built without prerequisites. This makes SIONlib a lightweight and easy-to-install library.

SIONlib is implemented in the programming language C. The SIONlib C 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 file only requires replacing the open and close calls, as we will see below. 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 use the library, a FORTRAN language mapping is supplied in addition to the C API. Multi-files with multiple underlying physical files are handled transparently. The next sections will discuss the following four modes of accessing a SIONlib's shared file container: *parallel write*, *parallel read*, *serial write*, and *serial read*.

3.7.1 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 (cf. Listing 3.1). Besides file name and open mode, the open call takes the maximum number of bytes expected to be written in one piece (`chunksize`) as a parameter, which can be individually chosen for each task. The global communicator `gcom` includes all tasks for which a logical file needs to be created. The

```
sid=sion_paropen_mpi( fname, "bw",              /* open, collective */
                      &numfiles, gcom, &lcom,
                      &chunksize,
                      &fsbsize, &gblrank,
                      &fileptr, &nfname);
                                                /* write, non-collective */
sion_ensure_free_space(sid, nbytes);            /*     with ANSI-C call */
fwrite(data, 1, nbytes, fileptr);

sion_fwrite(data, 1, nbytes, sid);              /*     or, with SIONlib call */

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

Listing 3.1: SIONlib Parallel write.

parameter numfiles and the local communicator lcom can be used to define a subset of tasks that share the same underlying physical file. Further parameters are the size of a file-system block (fsbsize), which can be used to specify a non-default block size, the global rank of the tasks (gblrank), and in case of multi-files, the filename of the physical file used by this task (*nfname*). The open operation returns two file handles: (i) a normal ANSI C file handle for the task-local file to be used in subsequent ANSI C write operations just as if the logical file was a physical file and (ii) a SIONlib file handle to be used in subsequent calls to the SIONlib API. For most of the parameters the corresponding pointer is passed to SIONlib's open call (call-by-reference). This allows the definition of only one open function for write and read mode. In write mode, all parameters have to be initialized and to contain values, whereas in read mode these parameters are filled by the open function with information from the file metadata block. Writing can be done in two ways, either by using the standard ANSI-C fwrite function or by using the SIONlib wrapper function sion_fwrite. The function sion_ensure_free_space has only to be called in the first case, 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. sion_fwrite simplifies the handling of writing and opens the possibility for optimization, because in this case, file interaction is under the control of SIONlib and operations to update the internal status information about the file are not needed. 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. In addition, all three options can be mixed freely. This applies to the other access modes accordingly.

3.7.2 Parallel read

Reading the multi-file in parallel is similar to writing it (cf. Listing 3.2). Again, open and close are collective operations, whereas the actual reading can occur individually. A call to sion_feof ensures that the end of the file has not yet been reached, similar to the standard feof call for ANSI-C. 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 with the customized read function sion_fread.

```
sid=sion_paropen_mpi( fname, "br",              /* open, collective */
                      ... &fileptr, ...);

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

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

Listing 3.2: SIONlib parallel read.

3.7.3 Serial write

In addition to writing a SIONlib file from a parallel application, the serial API also offers functions to write a SIONlib file from a serial application (cf. Listing 3.3), a necessary prerequisite to build serial pre- and post-processing tools. Since only one process executes the open call now, a whole array of chunk sizes needs to be supplied as a parameter. The `sion_seek` call helps to navigate within the multi-file, allowing the user to conveniently locate a specific position within a given chunk of a given task (e.g., the task with rank *i*).

```
sid=sion_open( ..., &chunksizes, &fileptr); /* open, serial */

for(rank=0;rank<size;rank++) {             /* loop over ranks */
   sion_seek(sid, rank, chunk, pos);       /*   seek chunk */
   sion_fwrite(..., sid);                  /*   write data */
}
sion_close(sid);                           /* close, serial */
```

Listing 3.3: SIONlib serial write.

3.7.4 Serial read

Serial reading can happen with either 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 multi-file in the local-view mode, the rank of the task is supplied as an argument to the open operation (cf. Listing 3.4). The actual reading is done in the same way as in the parallel case.

```
sid=sion_open_rank( ..., rank, &fileptr);   /* open one rank, serial */

/* reading like in the parallel case */

sion_close(sid);                            /* close, serial */
```

Listing 3.4: SIONlib serial read with task-local view.

If a SIONlib file is opened in the global view mode (cf. Listing 3.5), the user usually needs to retrieve first all metadata to obtain 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);               /* open, serial */

sion_get_locations(sid, ...,                /* get file info */
                   &nrranks, &nrchunks,
                   &chunksizes, ...);

sion_seek(sid, rank, chunk, pos);           /* seek chunk and position */
fread(..., fileptr);                        /* read data */

sion_close(sid);                            /* close, serial */
```

Listing 3.5: SIONlib serial read with global view.

3.7.5 Fortran interface

Taking into account that numerous scientific codes are written in Fortran, SIONlib provides Fortran language bindings. 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. This is needed, because Fortran cannot write to ANSI C or Fortran file handles directly. Instead, Fortran provides write and read functions for unformatted I/O, which accept a Fortran unit number as file handle. Furthermore, Fortran adds a record structure to the data in the binary file, which would affect SIONlib's internal file structure. With the SIONlib wrapper function, data is written unmodified to the file, which also allows data to be read later from programs that are implement in other languages.

3.7.6 Commandline Utilities

Standard Unix tools cannot read or modify SIONlib files directly, because these tools cannot interpret the file format. Therefore, SIONlib provides a set of serial commandline utilities to work with SIONlib file containers. All these tools use the serial API of SIONlib and are therefore examples of how this API can be used to build customized tools.

The dump tool `siondump` prints the metadata of the SIONlib file to standard output. This is a convenient way to learn more about the structure of the multi-file in order to see, for example, how many logical files it contains and how large they are. With additional options, the dump tool can generate reports on task level, for example, the chunk size, the filling rate, or the file number of a multi-file container (*task mapping*). The tool `sionsplit` extracts all distinct logical files from a given SIONlib file and recreates the corresponding physical files, whereas `sioncat` extracts only the data of one task from the SIONlib file and prints it to standard output or to a file. The defragmentation tool `siondefrag` generates a new multi-file from an existing one, contracting all of the chunks of a task, which are spread in the file over multiple blocks, into a single chunk. The new file contains only one chunk per task. In addition, all gaps in the form of unused file-system blocks are removed.

3.8 Coalescing I/O

SIONlib file containers can become very sparse, if the number of tasks is high and the individual data sizes are very small. As proposed in Section 3.2.3, SIONlib implements a aggregation of data among the tasks (*coalescing I/O*). The general strategy is illustrated in Figure 3.12. The application tasks are divided into two groups namely collectors and senders. The collector tasks receive data from the sender tasks and write their data on behalf of these to the file container. A sender task sends its data to the assigned collector instead of writing it to the file container.

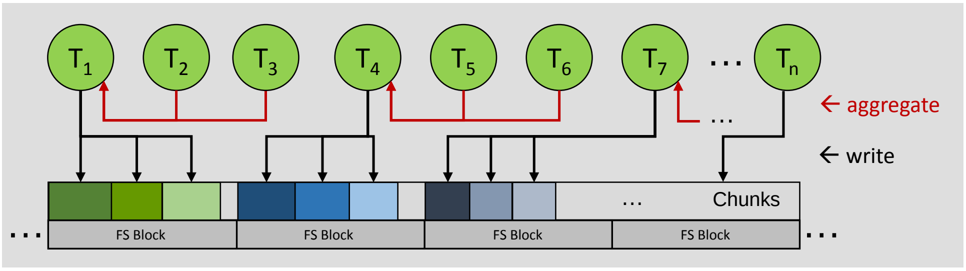


Figure 3.12: Simple example for coalescing I/O with SIONlib, which reduces the file size by a factor of three, assuming that data of at least three tasks fits into one file-system block.

The main advantage of this aggregation scheme is that the collector task can write larger contiguous blocks of data, because chunks written by one collector do not have to be aligned to the boundaries of file-system blocks. Furthermore, the size of the resulting SIONlib file is reduced by the size of the gaps between the sender chunks in the classical one-to-one scheme of SIONlib. Gaps in the file container only exist between the sets of data chunks that will be written by different collector tasks. Additionally, the coalescing approach leads to a reduced number of writer tasks and therefore fewer I/O streams, which potentially reduces the probability of congestion in the I/O infrastructure. In general, the distribution of collector tasks is variable, but cannot be changed after creating the file container. SIONlib knows the maximum size of the chunks in advance and is therefore able to compute a distribution scheme of collectors according to the given chunk sizes. The optimal distribution depends not only on the chunks. Additionally, the file-system block size, the structure of the I/O infrastructure and the characteristics of the interconnect are additional decisive factors. An additional factor is the file density, which results in a trade-off between I/O efficiency and disk space usage. Currently, SIONlib assigns so many tasks to a collector that a file-system block is almost be filled, whereas the number of tasks may not extend a certain limit. The number of sender tasks is computed individually for each collector, so that the number of sender tasks can vary across the collectors. Optionally, the user can set the number of collectors himself. In this case, sender tasks are distributed equally across the collectors.

Consequently, the write and read operations have to be collective, because communication is needed between sender and collector tasks. The SIONlib API provides special calls for this: `sion_fwrite_coll` and `sion_fread_coll`. In general, the data aggregation can be implemented with collective or with point-to-point communication patterns. The underlying communication pattern is not uniform in general, because chunk sizes do no have to be equal for all tasks and, consequently, the number of the assigned senders per collector can vary across the collectors. This makes an efficient implementation with collective communication operations difficult. Therefore, for hybrid applications the coalescing scheme implements a hierarchical aggregation: OpenMP threads aggregate their data first, and one of the threads will then participate in SIONlib's MPI coalescing scheme.

Particular attention has to be paid to the memory usage of the collector tasks, as they have not only to maintain their own data, but also to aggregate data of the sender tasks. SIONlib pursues the approach of minimizing the buffer space to the size of one file-system block. This is reasonable, because I/O operations have good performance if they operate on blocks of this size. On the other hand, coalescing I/O is only applicable to small chunk sizes, so that typically data of one collector will fit into one or a few file-system blocks. The collector loops over the sender task and collects the data task by task. Every time the receive buffer is filled, the collector flushes it to disk. The sender tasks send data to the collector only on request. In this way, the collector receives data only from one sender at a time. Overall, this communication scheme can be seen as a compromise between efficiency and memory usage.

3.9 Key-Value Containers

The size of a chunk in a SIONlib file container has a lower limit. To ensure the exclusive access to file-system blocks, a chunk has to have a least the size of one file-system block. With coalescing I/O, SIONlib can circumvent this issue for applications that can perform the write and read operations collectively. Other applications that are not able to perform collective I/O will not benefit from SIONlib's approach, because they have to allocate larger chunk sizes, although they need to write only a few bytes. Furthermore, applications and tools that do not know the exact number of participating tasks/threads in I/O operations in advance have similar requirements for a more flexible storage scheme.

To support I/O of fine-grained structured data with variable number of partitions, the design of SIONlib is enhanced by task-local *key-value containers*. Instead of raw data, tasks can now store sequences of short data blocks, each attributed with a key. Especially, applications with multi-level parallelization can use this feature by integrating SIONlib into the outer MPI parallelization level and storing thread-related data as key-value pairs in the task-local container. In this case the key would be the thread number. Pairs with the same key can occur multiple times in a chunk. In this way sequences of data blocks with the same key represent a virtual data-stream on key level.

For this purpose SIONlib's API provides a special open mode and a number of additional functions. The function `sion_fwrite_key` writes one key-value pair to the task-local chunk. The read function `sion_fread_key` requires a key as parameter, which is used to look up the next occurrence of data for this key and to read it from the chunk. Seeking in data blocks of the same key is possible with the `sion_seek_key` function, similar to the `sion_seek` function for normal data chunks. Additionally, SIONlib offers iterator functions to operate on SIONlib files with an unknown set of keys.

Only functions on the task-local level were added or modified to implement the key-value containers. The overall file container and the metadata handling did not have to be modified. The structure of the local key-value container can be defined in different ways. For example, descriptive information (e.g. key, size, or offset) can be stored in a separate, local metadata block or it can be interleaved with the data. The right selection of an implementation depends on different factors, like the overall data size, the size of a single data entry, the number of key-value pairs, and the number of different keys. Therefore, SIONlib implements an abstract API for key-value functions and users can select one of the implemented storage types at open time.

Currently SIONlib only supports the storage type *inline*, which interleaves descriptive data and application data as depicted in Figure 3.13. A short metadata block containing the key and the data size precedes each data block. A task that reads data for a certain key has to

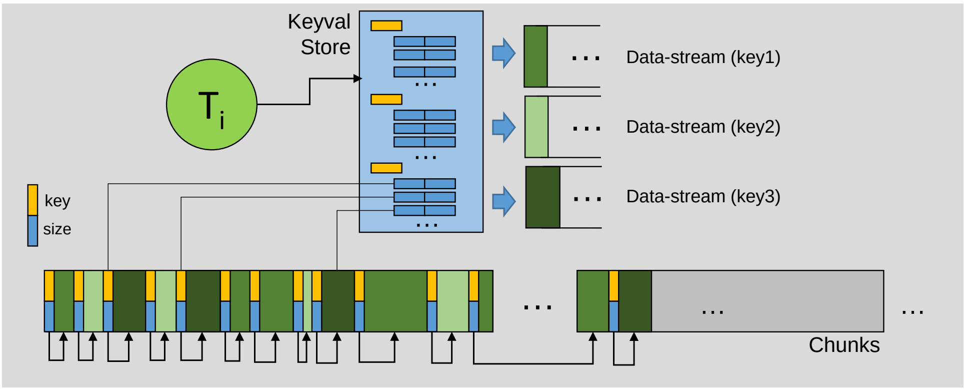


Figure 3.13: Simple example of a SIONlib key-value container using an *inline* storage scheme. The key and data size are interleaved with the data. Each task maintains a data structure in memory to buffer metadata of already read key-value pairs to optimize the procedure.

start at beginning of the file and has to traverse all metadata blocks until the block of the requested key is found. To optimize the read and look-up procedure, SIONlib maintains a hash table in memory, which contains key, size and offset for each key-value pair already found. As indicated in Figure 3.13, data blocks can be extended to the next chunk if space is not sufficient in the current chunk.

3.10 Support for Tools

Parallel tools like Scalasca [36] or Score-P [59] place different requirements on parallel I/O libraries. These tools interact with applications and inherit additional restrictions from them concerning the parallelization scheme and runtime configuration. Therefore, tools like Score-P have to support applications with different parallelization schemes, as they instrument the application code to obtain event-trace data.

The Score-P instrumentation and measurement infrastructure, which is used by Scalasca and Vampir, provides support for applications with a variable and not pre-defined number of threads and hence, the I/O layer, which is used to store event traces on disk, must support this feature as well. SIONlib fulfills this requirement by providing the key-value container. For hybrid applications, Score-P records event traces on each thread, but writes the event traces to output files from only one thread. The output format of these files is the OTF2 format. Conversely, VampirServer, the parallel version of Vampir, and Scalasca read OTF2 data from multiple threads concurrently. Therefore, SIONlib provides a function to duplicate a SIONlib file handle (`sion_dup`) as an additional enhancement. Once a SIONlib file is opened in parallel, this function can be used to replicate the internal data structures for each thread. This allows each thread to perform independent read operations. Further tool-specific support is available in SIONlib with the generic API, with the reinitialization of already opened files, and with the *mapped* parallel open mode, which supports to open SIONlib files with a different number of tasks at file creation time.

3.10.1 Generic API

The generic interface of SIONlib is primarily designed for the integration of SIONlib into the Score-P infrastructure. Score-P internally uses an abstract communication layer and provides callback functions to propagate the communication methods to underlying software layers. As described in Section 3.7, the interfaces of SIONlib for MPI, OpenMP, and hybrid applications provide communication methods via callback functions to the generic parallel layer in the same way. SIONlib only needs a few communication functions for metadata management in the parallel generic layer. These are the *broadcast*, the *gather*, and the *scatter* method to distribute and collect metadata, as well as a *barrier* method to synchronize the tasks. All methods are collective and have to be performed either on all tasks or on a subset of tasks. The global communicator group consists of all tasks participating in SIONlib I/O, whereas the local communicator group includes only those tasks that access the same physical file.

SIONlib's generic interface provides an API to define and register a user-defined set of callback functions and helps in this way to implement a new parallel API for SIONlib, for example, to support applications and tools that are based on parallelization paradigms other than MPI or OpenMP (cf. Figure 3.11). This has simplified the integration of SIONlib, because the Score-P callback functions can be propagated directly to the generic interface of SIONlib. A more practical advantage of the generic interface is that all required software layers have no dependencies to external libraries. This eases the integration of SIONlib into the build environment of tools.

3.10.2 Reinit

The `reinit` function of SIONlib allows postponing the exact specification of chunk sizes to a later time of execution, which is possible as long as no data has been written to the file. Reinit was implemented as a special feature for Scalasca 1.x to improve the efficiency of SIONlib I/O for trace data. Scalasca provides an internal buffer on each task, which is filled with trace data during runtime. Typically, the buffer is large enough to store all data of the run and it will be written to a SIONlib file at end of execution. Only in the rare cases of insufficient free buffer space, the buffers have to be flushed during runtime as illustrated in Figure 3.14 (left). Therefore, the file has to be created and opened at start time to be prepared for intermediate flushes. However, at this time, the exact chunk size is not known and Scalasca has to specify the size of the memory buffer as a maximum chunk size. In the case that the memory buffer size is much larger than the size of the recorded event set the resulting SIONlib file would be very sparse and I/O would be less efficient, because file systems blocks are not filled completely. The exact chunk size is known at the end of the execution and can be passed to SIONlib via the `reinit` call. SIONlib will then reinitialize the internal data structures without recreating and reopening the physical files. This feature becomes more important, if the data in the memory buffer is compressed in-place and not during the write operation as it is implemented in Scalasca (1.x). In this case, the chunk sizes can be further reduced to the size of the compressed data. Especially in combination with the coalescing I/O feature of SIONlib, files can be much denser as depicted in Figure 3.14 (right).

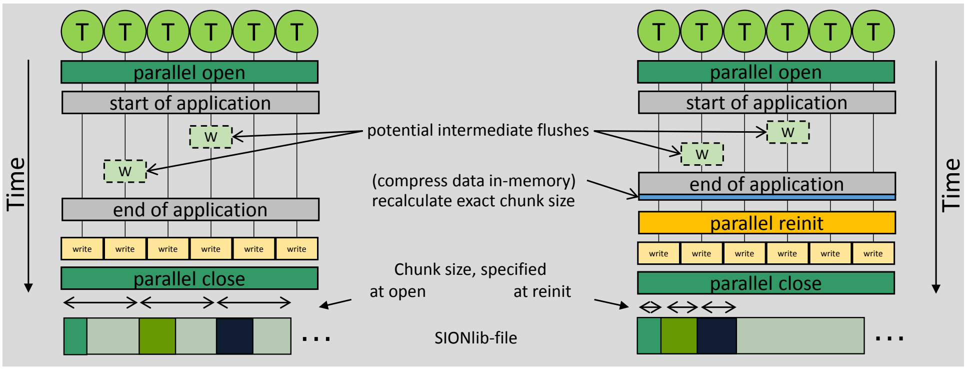


Figure 3.14: Example of using the SIONlib reinit feature in Scalasca. The instrumented application records event traces in memory buffers. The use of the SIONlib reinit function reduces the chunk size of the forehandedly opened file to the actual required sizes (right) compared to the original scheme (left).

3.10.3 Mapped open

Parallel access to SIONlib files requires that the number of participating tasks is equal to the number of chunks in the file. Therefore, applications reading a previously created SIONlib file have to run with the same size as the creating application. Applications or parallel tools that run with less or more tasks can only fall back to the serial API to open the file container on each task individually. As the memory and logistic overhead for this serial open is very high compared to the parallel collective open, the feature *mapped open* was added to SIONlib. This feature is not only useful for restarting applications on a different number of tasks. It also supports parallel tools that often run in smaller configurations than the simulation itself. Parallel post-processing tools are becoming more important as the size of application output files grows in such a way that they cannot be moved off-site and have to be processed at the same location. A special use case is VampirServer, the parallel version of the performance analysis tool Vampir. The server part of VampirServer is used to access large trace data directly on the HPC system, instead of moving it first to a local desktop system or reading it serialized on a login node. VampirServer interacts directly with a Vampir client, which is running on a local desktop system or a login node of the HPC system. As VampirServer runs in parallel, it can read and process trace data in parallel on the HPC system and therefore it provides a faster interaction with the user and requires less data movement than reading the data locally on a desktop system. However, VampirServer typically runs on a moderate number of tasks, which is mainly defined by the required main memory needed to store the trace data of large runs.

Figure 3.15 shows a simple example for a *mapped open*, where a SIONlib file is read from an application, running only with half the number of tasks. With *mapped open*, each task can specify an individual list of global rank numbers whose data chunks are intended to be read from this task. The *mapped open* is a collective operation. Therefore, the metadata of the SIONlib files is read only once from a file by one task and is distributed to the other tasks with collective communication operations, similar to the parallel open. Furthermore, SIONlib only has to maintain the metadata of the specified tasks in memory and only has to open those physical files that contain a requested chunk. In comparison with a native serial open from each task, the operation is more efficient in memory usage and causes less file-system activity.

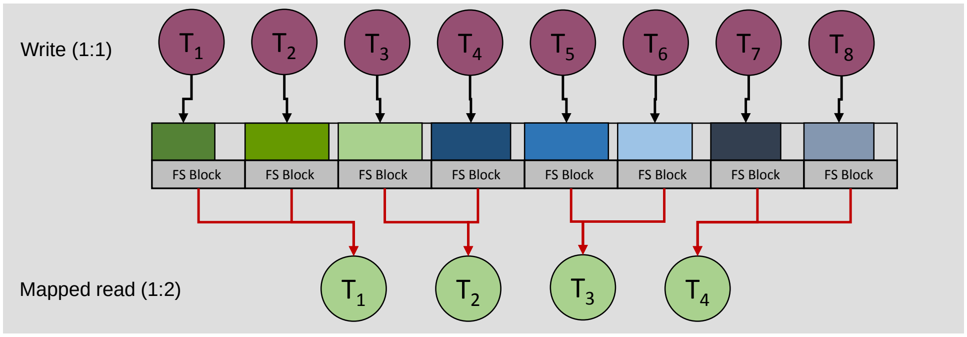


Figure 3.15: Example of using the SIONlib mapped open to read a file container with four reader tasks, while it was initially written using eight tasks. Each of this reader tasks now has to read chunks from multiple writer tasks (in this example two).

The *mapped open* also supports the creation of SIONlib files with a different number of tasks. This is typically used in parallel preprocessing tools to prepare input files for larger simulation runs.

3.11 Related Work

One of the strategic goals of parallel I/O libraries on higher abstraction level is to exploit knowledge of access patterns to optimize the data flow between applications and disks, utilizing the parallelism available on hardware and software layers in-between. Typically, this is done by introducing an additional software layer, which applications can use to describe data types, the data structure, and its distribution among the application tasks.

A prominent example of such a platform-independent interface supporting parallel binary I/O is MPI I/O [69]. 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. Each task can specify a non-contiguous view of a shared file, greatly simplifying the work with fined grained data distribution schemes. Apart from these more advanced features, MPI I/O also offers all mechanisms needed to perform I/O in the traditional way, similar to the POSIX I/O interface. While offering high-level functionality for striped and irregular access patterns, a transparent mapping of many logical task-local files on few physical files is not directly supported, although it can 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, MPI I/O can be seen as orthogonal to the SIONlib approach.

MPI I/O models data in terms of basic data types and user defined data types. Data structures are derived from these data types by specifying their locations in an address space. High-level parallel I/O libraries, like HDF5 [42], Parallel NetCDF [63], and NetCDF-4 [91], allow reading and writing of data in terms of structured data models including annotated multidimensional arrays of typed elements and hierarchical groups of objects. These libraries also store metadata describing the specific data format in addition to the actual data in order to facilitate easy sharing of files. The libraries support 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, as described in Section 1.2.1, high-level libraries store the data in a global file representation, which potentially requires an unnecessary data reordering, when output files are intended for checkpointing and restarting.

ADIOS [65] 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 high-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 fact, the ADIOS-BP 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 [51]. Second, a footer index ensures that the data section can grow beyond what is known at file creation time without moving data.

In addition to high-level parallel I/O libraries, where optimization is one of the goals besides the simplification of complex I/O operations on application-based data structures, one can find a number of approaches that propose strategies or implement libraries to optimize existing I/O solutions. For example, to optimize two-phase implementations of MPI collective I/O operations, Liao et al. [64] proposed dynamic file partitioning techniques that align the file domains assigned to aggregator processes with file-system block boundaries. This helps to avoid the serialization of I/O operations caused by lock conflicts when multiple aggregating processes want to write in parallel. The approach 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 aggregating processes.

TBON-FS [12], a virtual file system, defines scalable operations on whole groups of files. It allows a client to communicate efficiently 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 iterations across all group members, TBON-FS still operates on a potentially large number of physical files.

The I/O Forwarding Scalability Layer IOFSL [3] represents an approach to implement I/O forwarding for parallel applications over dedicated IOFSL servers, similar to I/O forwarding on IBM Blue Gene/Q systems. IOFSL provides a framework to implement strategies to exploit knowledge about application I/O patterns and to implement an optimized treatment of application I/O requests on the I/O server. For example, this strategy is used to support task-local I/O patterns of VampirTrace and Vampir at large scale by adding aggregation techniques to the IOFSL servers [48]. As IOFSL servers have to be deployed in user-space if IOFSL is not installed on file-system nodes or special I/O nodes, additional compute nodes have to be allocated to run applications together with the IOFSL framework. This requires a higher effort and increases the complexity of the application software environment, whereas SIONlib implements similar optimization strategies in a lightweight library without further requirements to the runtime environment.

Techniques like SIONlib's multi-file approach, which is used to reduce the metadata overhead by accessing shared files from a large number of tasks, are meanwhile already integrated into other I/O libraries. The technique of transparently dividing shared Parallel NetCDF files into multiple physical files is denoted as *subfiling* [34] or *file partitioning* [86]. Subfiling is integrated in the latest version of Parallel NetCDF (1.4.1). A similar technique to subfiling is also implemented in GLEAN [15], a topology-aware data movement and staging framework. This framework improves I/O performance by leveraging knowledge of the structure of the internal network and the I/O infrastructure. Further techniques of GLEAN are the adaptation of data movement in collective I/O operations to the underlying network topology and the reduction of data size via lossless data compression. The efficiency of the approach is shown using benchmarks and applications on a Blue Gene/Q system with the GPFS file system. The results demonstrate good efficiency when using subfiling with one file per I/O node, which is similar to SIONlib's multi-file approach for Blue Gene/Q.

The Parallel Log-structured File System PLFS [7] implements a user-based file system on top of the existing (parallel) file system. The goal of PLFS is to optimize parallel I/O for different usage patterns. Optimization modes are provided for shared file I/O, flat file I/O (task-local files), and small file I/O operations. Applications can access the file system via a patched MPI-library if they use MPI/IO. Alternatively, PLFS API calls can be integrated or the file system can be accessed by using the unmodified I/O call via the kernel module FUSE (Filesystem in Userspace), which may introduce additional high overhead [6]. For shared files, PLFS splits parallel I/O streams to a shared file into I/O streams to individual files, which are stored in a hierarchy of directories. This approach is diametral to the SIONlib approach as it increases the number of files and replaces shared file I/O with task-local I/O. PLFS can limit the metadata overhead for parallel file creation by distributing the file over a hierarchy of directories. By using individual files, PLFS omits the concurrent access of file-system blocks, which is caused by unaligned access. SIONlib solves this problem by implementing alignment is its shared file format to optimize the I/O. In contrast to PLFS, which uses the file system to store metadata collectively in a separate file, SIONlib can use the application's communication layer to aggregate and distribute metadata across the tasks without involving the underlying file system.

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 aggregated 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 [26, 80], the concurrent access to the file-system blocks that contain the directory inode more or less serializes this operation. SIONlib can handle this situation better only because it relies on superior knowledge of the intended access pattern.

4 Evaluation of Task-Local I/O with SIONlib

SIONlib is designed to support parallel task-local I/O at large scale. The performance of the approaches implemented in SIONlib is shown in this chapter with quantitative I/O benchmarks on the IBM Blue Gene/Q system JUQUEEN [53] at JSC and for different application use cases. JUQUEEN represents a large-scale system that requires efficient parallel I/O for its applications to run up to 1.8 million MPI tasks. Furthermore, it is an appropriate example for hierarchical I/O infrastructures, which will be seen more often in future exascale systems.

After discussing the architecture and I/O infrastructure of the IBM Blue Gene/Q system, results of I/O benchmarks on JUQUEEN will be shown from small scale (base line measurements) to the full system size with more than 1.8 million tasks. Furthermore, the impact of concurrent usage of shared resources like the file system on application I/O performance will be discussed. As an example, measurements demonstrating these effects are shown for the Lustre file system and the Cray XT system at Oak Ridge National Laboratory. The usability of SIONlib for checkpointing, which is one of the designated use cases, is shown with the simulation code MP2C for massively parallel multi-particle collision dynamics on JUQUEEN. Finally, tool support will be discussed as second use case of SIONlib. Results will be shown for the Scalasca performance tool set and SIONlib's integration in its measurement environment will be described.

4.1 Architecture and I/O-infrastructure of Blue Gene/Q

The IBM Blue Gene/Q system JUQUEEN has a modular structure as depicted in Figure 4.1. The compute nodes (CN) are equipped with an IBM PowerPC A2 CPU and 16 GiB of main memory. Applications have access to 16 cores of the CPU, a 17th core is used internally.

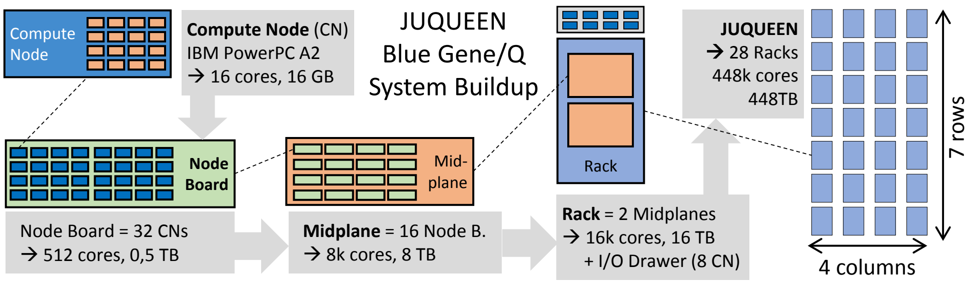


Figure 4.1: Components of IBM Blue Gene/Q system JUQUEEN.

The cores support 4-way SMT, which allows programs to run with up to 64 threads or tasks on one compute node. At least 2-way SMT is recommended to exploit the capability of the floating-point units. Thirty-two of these compute nodes build a node board (or node card). A node board with 512 cores or 2,048 possible threads is the smallest partition that can be allocated on the JUQUEEN system for user jobs. Sixteen of these node boards are packed into one midplane, which has a size of 8,192 cores (8k cores) and 8 TiB of main memory. Whereas jobs with a size of one or a few node boards are intended for test and development runs, production jobs should have a size of at least one midplane, which requires parallel applications to run with at least 16k tasks. A Blue Gene/Q rack consists of two midplanes. JUQUEEN has twenty-eight of these racks and provides therefore resources to run application with up to 1.8 million tasks. Additionally, a rack can host up to 32 I/O nodes (ION) in special I/O drawers. The IONs are equipped, similar to the CNs, with an IBM PowerPC A2 CPU and 16 GiB of main memory. Furthermore, they have access via PCIe (gen2) slots to a network interface (10GigE or Infiniband) [40]. Twenty-seven racks of JUQUEEN are equipped with 8 IONs each and one rack hosts 32 IONs.

The Blue Gene/Q system has a 5D torus network, connecting each of the CNs with its ten neighbors over links with a bi-directional bandwidth of 2 GiB/s. In contrast to the predecessor system Blue Gene/P, the torus network itself handles collective and barrier communication operations. As the smallest partition, the node card represents a 5D torus with a size of `2x2x2x2x2` CNs. The last dimension is limited to a size of two, in order to connect two CNs within the same node board. As the next scaling step, a midplane builds a torus of `4x4x4x4x2` CNs, a rack has a size of `4x4x4x8x2` CNs and the 5D torus on the full JUQUEEN system extends to `8x28x8x8x2` CNs. The core number on a CN will often be labeled as the sixth dimension of the torus. This extends the naming scheme of the torus dimensions to `ABCDE T`. Using this labeling scheme, applications can specify at program startup how application tasks should be mapped on the torus.

Whereas CNs are running a reduced Linux kernel, IONs and login nodes are running a full Linux operating system. One example for the limitation of the reduced kernel is that CNs have no support for direct access to the file systems. Therefore, the operating system on the CN cannot execute I/O system calls directly, instead, the I/O requests are forwarded to the IONs, which have access to the file system.

The JUQUEEN system is connected via the IONs to the Jülich Storage Cluster JUST, which hosts the GPFS home and scratch file systems (cf. Section 1.2.3). The scratch file system, on which the benchmark and application evaluation was performed, has a capacity of 3.8 PiB and provides a maximum aggregate I/O bandwidth of about 160 GiB/s to JUQUEEN. The IONs are connected via two 10 Gigabit Ethernet (10GigE) links to a 10GigE network switch fabric, which is also connected to the I/O servers of JUST.

4.1.1 I/O-Forwarding on Blue Gene/Q

As already described, the I/O requests on CNs are automatically forwarded to I/O daemons on the IONs. I/O forwarding is implemented on the level of system calls as function forwarding. In this way, the system will forward the corresponding POSIX I/O calls to the daemons on the

IONs. The daemons on the IONs execute the corresponding I/O operations on behalf of the tasks running on the CNs. As shown in Figure 4.1, CNs on a Blue Gene/Q rack have typically access to eight IONs. This results in an ION to CN ratio of 1:128. Therefore, each daemon has to serve I/O requests for up to 8192 tasks. One rack of JUQUEEN has a better ION to CN ratio of 1:32, as it is equipped with 32 IONs to support small jobs for testing and development. Because of the higher ION to CN ratio this rack is not used in this work for measurements that do not require all racks.

One major characteristic of I/O forwarding on Blue Gene/Q is that I/O streams between CNs and ION are interleaved into one I/O stream between ION and the file system. Because the IONs have to forward data from the CNs to the file system or vice versa, I/O node performance is directly related to the ability to buffer data. Therefore, to transfer data between ION and CNs the I/O daemons on the IONs maintain an internal memory buffer. Data to be transferred between ION and the file system is stored in an additional memory buffer, the GPFS page pool. These buffers occupy most of the main memory on the IONs.

The IONs are not directly integrated into the 5D torus of the Blue Gene/Q system. Instead, two CNs, the so-called *I/O-bridge nodes*, are connected via their eleventh network link to the two network adapters of the ION. In this way, the I/O-bridge nodes are responsible for routing network traffic between the IONs and the CNs assigned to the I/O-bridge node. The I/O traffic is managed on a lower network transportation level, so that computation on the bridging CNs is not influenced. The communication between ION and CN is handled with Remote Direct Memory Access (RDMA) with the help of the 17th compute-node core.

The mapping scheme of IONs is hard-wired in the system. Each ION has two links, which are connected to I/O-bridge nodes in different node boards. The scheme for a configuration with four IONs per midplane is illustrated in Figure 4.2. Because node boards have to be attached with exactly two links, only 4 of the 16 node boards have a wire to the IONs. Therefore, two I/O-bridge nodes are located in one node board and are responsible for routing I/O traffic of CNs in four node boards. This leads to an irregular mapping of CNs to bridge nodes, which is

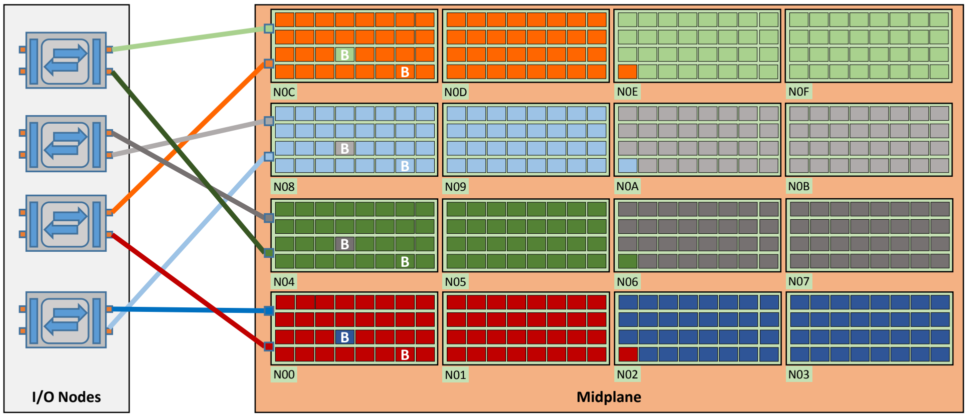


Figure 4.2: Mapping of I/O nodes to tasks of JUQUEEN BG/Q midplanes. The four I/O nodes have network links to eight I/O-bridge nodes (marked with 'B') that route I/O traffic of 64 compute nodes each (indicated by color mapping).

indicated by the color of the CNs in Figure 4.2. All CNs of a node board route their I/O traffic through the same I/O-bridge node, except the second I/O-bridge node itself (e.g. the blue CN in N00) which switches roles with one CN of another node board (e.g. the red CN in N02). The reason for this is that I/O bridge nodes always route their own traffic and the traffic of the CNs of the associated node boards. Therefore, to avoid an imbalance in I/O load, the second I/O-bridge node switches its assignment with one CN of another node board to guarantee the same number of CNs per I/O-bridge node.

In the JUQUEEN configuration, the IONs are connected to the GPFS file system with an *ethernet channel* over two 10GigE connections. The raw bandwidth on this channel is about 2.5 GiB/s, whereas the two Blue Gene/Q torus links together provide a raw bandwidth of 4 GiB/s. Consequently, the I/O bandwidth for an application has an upper limit of 2.5 GiB/s per ION. The measurement of the achievable bandwidth over one ION is subject of the baseline benchmarks, which will be discussed in Section 4.2.2.

4.2 I/O Benchmarks

Before testing SIONlib with applications, the I/O performance of the library has been evaluated with a benchmark program. This allows measuring different features of SIONlib under well defined conditions. The benchmarks are run in the previously described test environment of JUQUEEN and use the GPFS scratch file system on JUST (cf. Section 1.2.3). Furthermore, all runs were performed within the normal production environment and production time. This means that influences from external applications and activities on the shared resources network and file system could not be precluded. Therefore, most of the benchmark runs were repeated at least three times and the best results are reported here. Only a few large-scale and very costly runs were not repeated and were ran only once on the system. SIONlib has its own parallel benchmark program partest, which is designed to evaluate the I/O performance of shared file I/O with SIONlib at large scale and to compare the performance of shared file I/O and traditional task-local I/O with individual files.

First, the results of measuring the time for shared file creation with SIONlib will be presented and compared to the creation time of task-local files. Next, baseline measurements on one ION were performed with a quasi-standard I/O benchmark for this purpose, the IOR (Interleaved Or Random) parallel I/O benchmark [50, 82]. Results of I/O benchmark runs with IOR using shared-file I/O are compared with corresponding SIONlib results, to validate the benchmark program partest of SIONlib. The measurements, how the alignment of data chunks to file-system blocks optimizes the I/O efficiency, will be discussed as next, followed by measurements showing the limitations of shared file I/O to one physical file on JUQUEEN and the GPFS file system and how they can be avoided with the multi-file approach of SIONlib. Benchmark runs at large scale, up to 1.8 millions tasks, are performed solely with SIONlib and are shown for large data sizes using the standard configuration of SIONlib and for small data sizes using the coalescing I/O feature of SIONlib.

Write and read timings are measured in the following benchmarks as the time span between the first task starting to open the file and the last task completing the close operation of the file.

This means that the file creation as part of the open operation is accounted in the calculation of the write and read I/O bandwidths. Data sizes are specified in powers of two: e.g. 1 MiB = 2^{20} bytes = 1024^2 bytes = 1,048,576 bytes.

4.2.1 Parallel file creation

One of the major limitations of parallel task-local I/O is the handling of the large number of individual files. As demonstrated in Section 2.2.1, the time for the parallel creation of individual files in the same directory increases on GPFS linearly with the number of files, which is caused by the serialization of the directory i-node updates needed for each new files. SIONlib file-creation and file-open strategy is different to the traditional approach. Shared files are first created by one task and subsequently opened by all tasks. As result, only a few shared files have to be created and the time for creating the files reduces dramatically. Opening an existing file is not critical, because the directory i-node will not be modified by this operation. Therefore, the time for opening a file is independent of the number of tasks involved. Figure 4.3 shows the results of a measurement with partest for creating and opening a SIONlib file container on JUQUEEN. At the largest scale with 1.8 million tasks, SIONlib needs not more than 3 seconds for this operation. In comparison, creating individual files on JUQUEEN for this scale took about 777 seconds (cf. Figure 2.4 on page 32).

SIONlib adds some administrative overhead in its open function, because tasks have to be synchronized and metadata has to be exchanged. This shows up as additional time in the measurements. As recommended, the benchmark was configured for these measurements to use one file per I/O-bridge node. In this case, SIONlib's open operation gets as second parameter a local MPI communicator. SIONlib will create one shared file for each local communicator and will assign all tasks of the local communicator to this shared file. For convenience, the runtime environment on Blue Gene/Q provides the `MPIX_Pset_diff_comm_create` call as an extension to MPI, which returns exactly such a communicator, where all tasks of the same

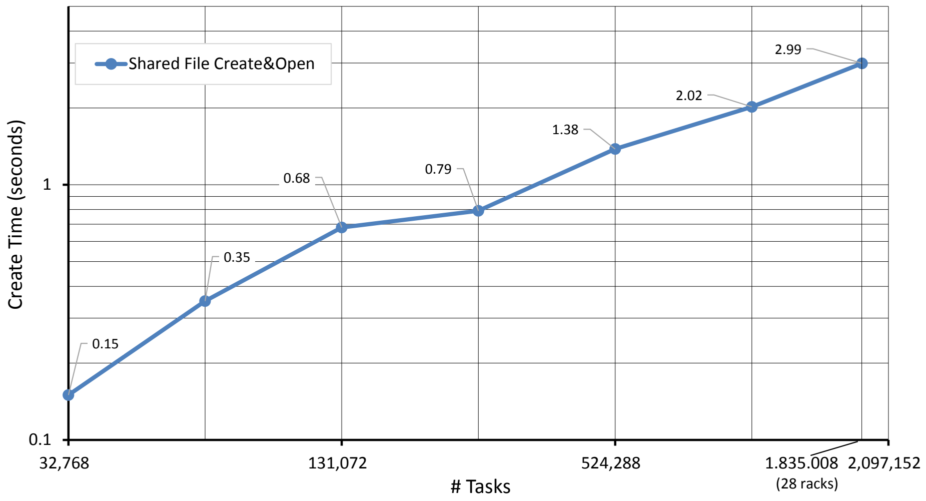


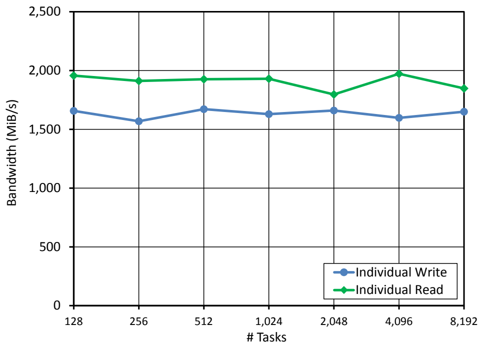
Figure 4.3: Creating and opening SIONlib container file in parallel on JUQUEEN with one file per I/O-bridge node.

communicator are routing their I/O traffic via the same I/O-bridge node. SIONlib has to perform only a few collective operations on the global communicator during file opening. For example, the first task assigns each local communicator a unique file number and communicates this to all tasks. Subsequently, collective operations for collecting or distributing metadata are performed on local communicators of each physical file individually. The slightly increasing time for opening the file shown in Figure 4.3 is partly due to the initial communication on the global communicator. Another reason is the impact of the I/O operations on the file system when the shared files are created. The number of physical files increases linearly with the number of I/O-bridge nodes to 495 files at largest scale. These files are created concurrently by the first task of each local communicator. In addition, at startup each of these tasks has to write the first metadata block to the file, which is also accounted for the time for creating and opening a SIONlib file. Overall, the time for opening and creating the files is significantly reduced and is negligible in comparison to the traditional approach with individual files.

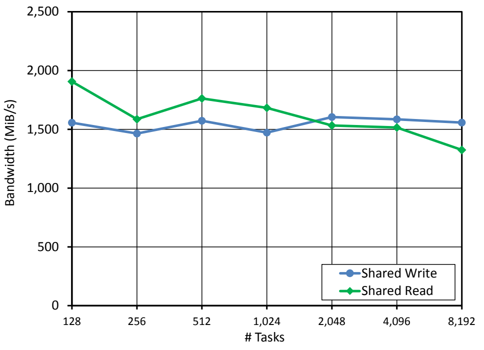
4.2.2 Baseline measurements

To evaluate the performance of SIONlib, we first discuss the results of baseline measurements on one ION of JUQUEEN with IOR and compare these with results using SIONlib. The results of baseline measurements help to assess the I/O bandwidth at larger scale in the subsequent measurements in this chapter. The scales of benchmark runs for baseline measurements were selected based on the criterion that they should run only one component of the I/O infrastructure. Because the I/O infrastructure separates I/O streams by assigning the CNs to different IONs, the smallest scale for a baseline measurement is in this way a partition that includes only tasks of one ION. However, as illustrated in Figure 4.2, the 128 CNs that are assigned to the first ION are not in a continuous partition. Moreover, they are located in five different node boards. To solve this, the benchmark was configured to run on a full midplane and the tasks had to be reordered with an own mapping file to select only the CNs that are assigned to the first ION. The remaining CNs of the midplane were kept idle during measurement. As only one ION is involved in the measurements, I/O bandwidth is naturally limited by the I/O bandwidth of this ION, which is about 2.5 GiB/s due to the two 10GigE external connections.

Figure 4.4a shows the results of measurements with the standard I/O benchmark IOR on 128 CNs with one file per task (individual file). The number of MPI tasks per CN is scaled from one to 64 tasks, so that the total number of tasks has a range from 128 to 8192 tasks. The data size per CN was fixed to 2 GiB of data, which results in an overall file size of 256 GiB. This amount of data exceeds the page pool size of 8 GiB on the ION by far and prevents the influence of cache effects in the measurements. The transfer size was set to 4 MiB, which matches the size of one file-system block. This guarantees that all I/O operations are aligned to the size of file-system blocks and that the runtime system and GPFS daemons can handle the I/O operations efficiently. The results can therefore be seen as an upper limit for real I/O patterns. Running the IOR with individual files the average write bandwidth is about 1,630 MiB/s and the average read bandwidth is about 16 % higher (1,900 MiB/s). As expected, these I/O bandwidths are independent from the number of tasks as they are achieved by independent I/O streams. Figure 4.4b presents the results of IOR with a similar configuration only that the benchmark uses a single shared file instead of individual files. In this configuration, the



(a) IOR using individual files

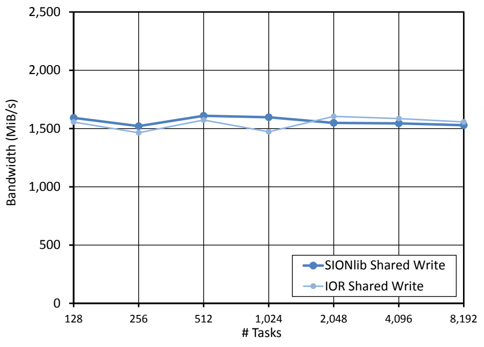


(b) IOR using one shared file

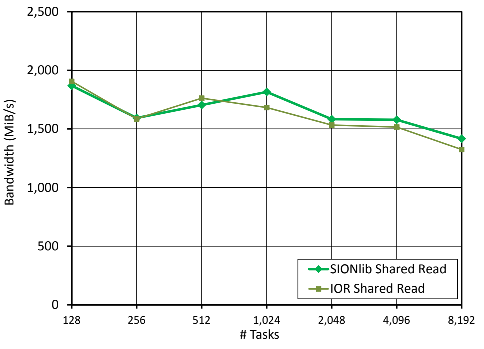
Figure 4.4: Baseline measurement with tasks using one I/O node on JUQUEEN with IOR (1-64 tasks per node, 2 GiB file data per node).

write I/O bandwidth decreases slightly to an average of 1544 MiB/s and is again not dependent on the number of tasks. However, the read I/O bandwidth decreases from 1,905 MiB/s at 128 tasks to 1,324 MiB/s at 8k tasks. Therefore, the handling of read requests depends on the number of tasks on the Blue Gene/Q system with GPFS, which indicates that these requests are handled differently than the write requests. Like the SIONlib benchmark partest, IOR computes the achieved bandwidth by dividing the number of bytes transferred by the time span between the first tasks starting to open a shared or individual file and the time the last task has finished the close operation.

Similar measurements were performed with the SIONlib benchmark program `partest`, using one shared file. Figure 4.5 shows the results in comparison with the IOR results and indicates that SIONlib can achieve similar I/O bandwidths, although SIONlib has a higher overhead due to the collective metadata handling and writing or reading of two additional metadata blocks. Similar to the IOR measurements, the read bandwidth with SIONlib decreases in the same way (cf. Figure 4.5b).



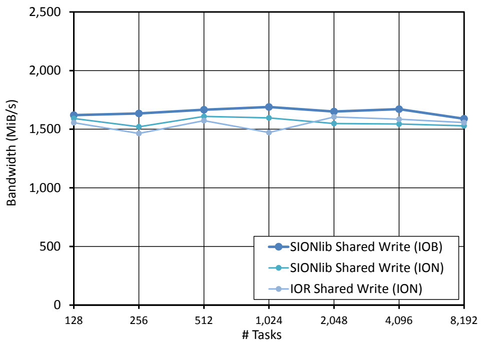
(a) Writing data



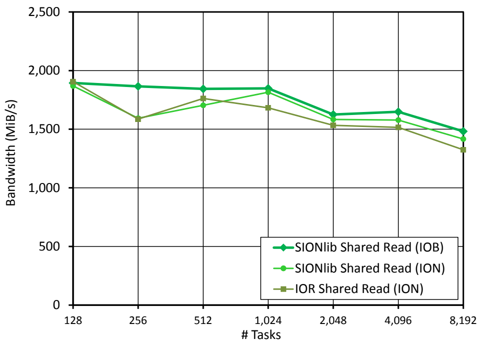
(b) Reading data

Figure 4.5: Comparison of IOR and SIONlib for baseline measurement with tasks using one I/O node on JUQUEEN (1-64 tasks per node, 2 GiB file data per node).

In a second benchmark, SIONlib was configured to use one file per I/O-bridge node, which yields in this case to two physical files, each accessed by 64 CNs. The results of the measurement are shown in Figure 4.6. The average write I/O bandwidth increases with this approach by 5 % from 1,562 MiB/s for one file per ION to 1,645 MiB/s for one file per I/O-bridge node. Since in both cases file metadata handling remains inside the ION and is performed by the same GPFS daemon, the observed I/O bandwidth improvement may be a result of a better handling of I/O streams and less overhead in local metadata handling on the ION. For example, file-system locks have to be negotiated only between 4k tasks for each of the files, instead of 8k tasks in the case of one file per ION. Assuming an exponential increase of the time for lock handling, it would be advantageous to use less tasks per file. Justified with the results of this measurement, most of the following benchmarks are performed in the configuration with one file per I/O-bridge node. This is also the recommended default configuration for applications using SIONlib on JUQUEEN.



(a) Writing data



(b) Reading data

Figure 4.6: Comparison of IOR and SIONlib, including the SIONlib's multi-file approach (IOB) for baseline measurement with tasks using one I/O node on JUQUEEN (1-64 tasks per node, 2 GiB file data per node).

4.2.3 Alignment and file locking

A fundamental strategy of SIONlib is to consider the size of the file-system blocks in the file-layout of the SIONlib shared file container (cf. Figure 3.8 on page 55). This is necessary, because the partitioning of a file space into file-system blocks has two implications for shared file I/O. First, the file system will only handle whole file-system blocks on the I/O-subsystem level and not the data chunks, which have a user-defined size. Second, file locking is, at least in GPFS, limited to the granularity of file-system blocks. Therefore, SIONlib aligns the boundaries between data chunks of different writers to the boundaries of file-system blocks in the shared file container.

The following measurements on JUQUEEN, using the scratch file system on JUST, demonstrate that such an alignment to file system block boundaries can avoid the degradation of the I/O write bandwidth caused by concurrent access of different tasks to one file-system block (cf. Section 3.2.1). The measurements were performed on one rack of JUQUEEN (1024 CNs) with the SIONlib benchmark program partest in a multi-file configuration with one file

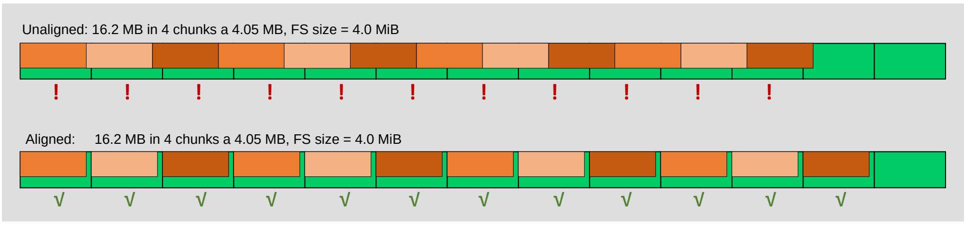

Unaligned: 16.2 MB in 4 chunks a 4.05 MB, FS size = 4.0 MiB
!
Aligned: 16.2 MB in 4 chunks a 4.05 MB, FS size = 4.0 MiB
√

Figure 4.7: Unaligned vs. aligned shared access (three tasks). Without alignment, all chunks are consecutively positioned in an interleaving order. As a result, all file-system blocks are accessed by two different tasks concurrently. In contrast, if alignment is enabled, file-system blocks are accessed by only one task.

per I/O-bridge node, which yields 16 physical files in total. The tasks were configured to write 16.2 millions bytes into the shared file container, and the data was separated into four chunks of 4.05 million bytes. This configuration was selected to fill the page pool on the IONs sufficiently and to avoid caching effects in the measurements. To evaluate the influence of alignment, the test was run once with the correct file system block size of 4 MiB and once by instructing SIONlib to use a much smaller file-system block size of 16 KiB, which disables the alignment in the file container. Figure 4.7 illustrates the resulting file layout of both tests for an example with three tasks. The alignment ensures that only one task accesses a file-system block. Without alignment, GPFS will serialize the access of two tasks to a file-system block. Furthermore, a file-system block can potentially be purged out of the page pool before the second task can write data to the block. Both effects reduce the I/O write performance as shown in Figure 4.8: using relatively small data size and aligning chunks to file system block boundaries, the I/O write bandwidth increases linearly with the number of tasks per ION from 4.0 GiB/s to 9.7 GiB/s using 64 tasks per CN. In contrast, the I/O bandwidth of the unaligned

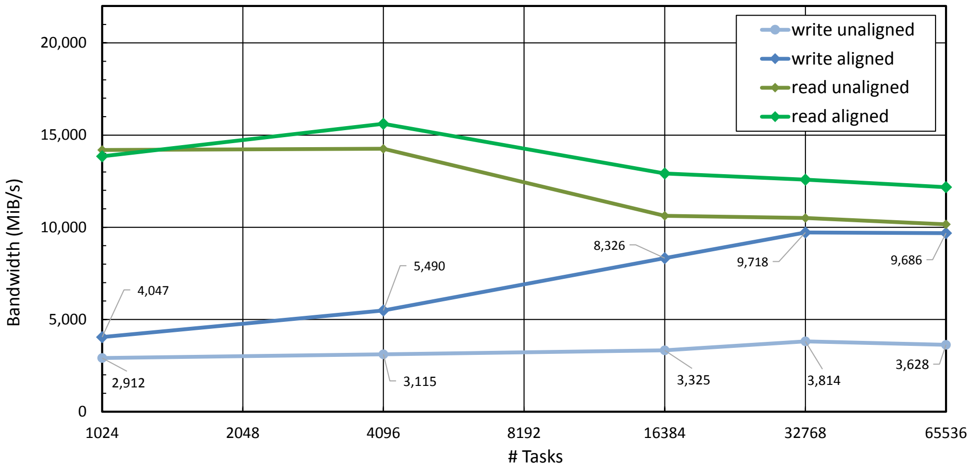


Figure 4.8: Measurement on JUQUEEN of aligned versus unaligned shared access, showing that alignment to file-system blocks increases the I/O bandwidth for writing significantly (1024 compute nodes with 1-64 tasks per node, 16.2 million bytes per task).

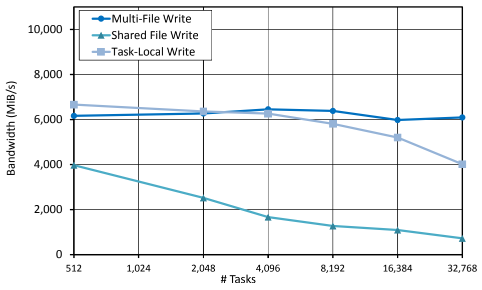
access does not exceed 3.8 GiB/s. Additionally, the measurements show that data reading is not affected by alignment in the same way. According to the baseline measurements, the read bandwidth is higher than the write bandwidth, but it is limited to a maximum of 2 GiB/s per ION. Therefore, the read bandwidth cannot exceed 16 GiB/s using eight IONs in the current measurement. With an increasing number of tasks, the read bandwidth increases first to its maximum value reached with four tasks per CN node and decreases then to its minimum value, using 64 tasks per CN. The reason for the first increase could be the read-ahead optimization of GPFS, whereas the decrease is according to the achievable read bandwidth in the baseline measurement on one ION (cf. Figure 4.4b).

These measurements show that the alignment to file-system blocks is essential. With such an alignment, high efficiency can be achieved, when the file-system blocks are filled more than half. For lower fill rates using the unaligned mode, a file-system block can store two or more chunks that are written by different tasks. As long as such blocks are not purged out of the page pool, such unaligned access can benefit from the dense packaging of chunks in file-system blocks, although the benefit will be diminished by the serialization of concurrent accesses due to locking. In aligned mode, GPFS has to manage full file-system blocks internally, even though those blocks are mostly empty. Therefore, applications should use the coalescing I/O feature of SIONlib for small chunk sizes to reduce the number of alignment points in the shared file and to benefit from the internal alignment of chunks in the SIONlib file container (cf. Section 3.8 on page 62).

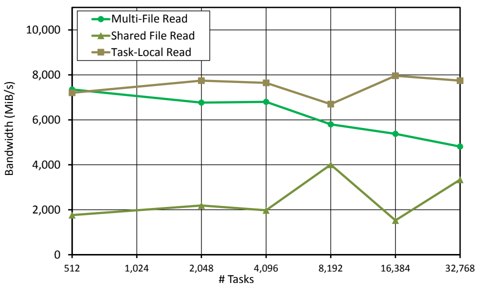
4.2.4 Shared file I/O with SIONlib

As discussed in Section 3.2.2, the number of tasks writing to one shared file is becoming a bottleneck at larger scale. SIONlib supports therefore shared-file I/O to multiple physical files. The comparison of this special SIONlib feature with the standard shared-file I/O will be discussed in this section. The results of the measurements on JUQUEEN are shown in Figure 4.9 for one midplane (512 CNs) and one rack (1024 CNs). Similar to the baseline measurements, the number of tasks per CN was scaled from 1 to 64 tasks, which leads to maximum size of 64k MPI tasks in this measurement with the SIONlib benchmark partest.

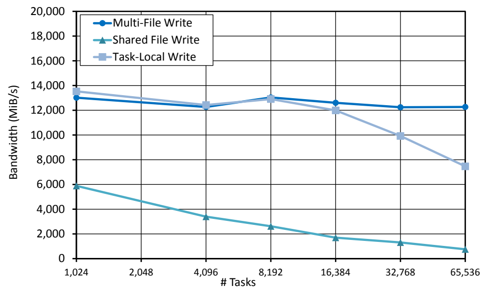
The first set of tests was run on one midplane using four IONs to handle the I/O streams. According to the baseline measurements on one ION and under the assumption that I/O to one physical file can be performed locally on the corresponding ION, the maximum achievable I/O bandwidth should be four times the I/O bandwidth of one ION (4*1645 MiB/s = 6580 MiB/s). The measurements show that using SIONlib with multi-file support can achieve an average write bandwidth of 6223 MiB/s, which is about 95 % of the computed I/O bandwidth (Figure 4.9a). The results of scaling these runs to one rack indicate that this bandwidth is nearly constant (95.5 % on eight IONs, Figure 4.9c). Furthermore, the read bandwidth of SIONlib with multi-file support also scales with the number of IONs involved and is about four times higher than the corresponding baseline measurements (Figure 4.9b and 4.9d). The achieved read bandwidth is lower than the read bandwidth of I/O from individual files, because of the additional metadata overhead in SIONlib and the Blue Gene/Q specific degradation of read bandwidth from a shared file as shown in the baseline measurements (cf. Figure 4.4b on page 77).



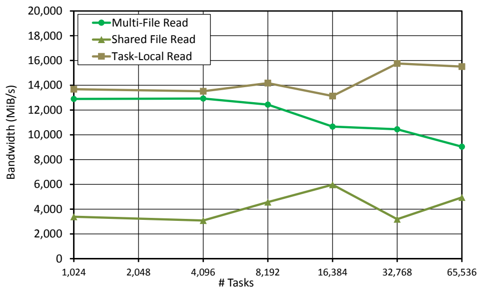
(a) Writing data (Midplane)



(b) Reading data (Midplane)



(c) Writing data (Rack)



(d) Reading data (Rack)

Figure 4.9: Comparison of SIONlib multi-file I/O with single-file I/O and with parallel task-local I/O on a midplane and on a rack of JUQUEEN (1-64 tasks per node, 512 MiB file data per node).

The I/O bandwidth of writing to individual task-local files decreases with the number of tasks. The reason is, that file creation, which is part of the file open operation, becomes more time consuming, whereas the time for writing remains constant. For example, with 64k tasks, 80 % of the I/O time was spent in file creation. As the files in the read measurements already exist, reading from individual files is not affected (Figure 4.9b and 4.9d).

As a comparison, the I/O measurements were also run with one shared file (cf. Figure 4.9, *Shared File Write/Read*). The results indicate that collective I/O over multiple IONs decreases the write and read bandwidth significantly. For example, the I/O bandwidth for writing is decreasing linearly with the number of tasks to a value below the I/O bandwidth of one ION at the largest scale. This demonstrates apparently that within this configuration (Blue Gene/Q systems and GPFS) parallel I/O to one shared file cannot exploit the available I/O bandwidth. A substantial bandwidth degradation can already be seen at a scale of one midplane or rack of JUQUEEN. Therefore, the remaining benchmarks at large scale were only performed with SIONlib's multi-file approach, which is also SIONlib's default configuration on such systems.

4.2.5 Scalability

Large-scale benchmark runs were performed with SIONlib up to the full system size of JUQUEEN (1.8 million tasks on 28 racks). Because of the limited scalability of traditional parallel task-local I/O and shared file I/O to one file, only the multi-file approach of SIONlib was used with one file per I/O-bridge node. The benchmarks were performed on JUQUEEN for different numbers of racks and each test was repeated with 1 to 64 tasks per CN. The data size per CN was configured to 256 MiB, which yields in an overall file size of 14 TiB on 28 racks. At full scale this data is stored in 496 physical files. In this case, the tasks running on the 27 racks that are equipped with 8 IONs (16 I/O-bridge nodes) have created 432 files. The task running on the special rack that is equipped with 32 IONs (64 I/O-bridge nodes) have created the remaining 64 files.

Figure 4.10 shows the I/O bandwidths for writing. As already observed in the measurements of the small-scale benchmarks, the I/O bandwidth for a given number of racks is constant for different numbers of tasks per CN. The slight variation at large scale (with more than 8 racks) is caused by the fact that the maximum of the usable file system bandwidth is reached and the tests become affected by other activities on the file system. The tests on 28 racks demonstrate the scalability of SIONlib's multi-file approach for data output even up to 1.8 million tasks. The results of the corresponding read benchmarks are shown in Figure 4.11. The trend of a decreasing I/O read bandwidth in the baseline measurements on JUQUEEN is also observed at larger scale and leads to a proportional drop of the I/O bandwidth from initially 150 GiB/s to about 85 GiB/s on 28 racks with 64 tasks per CN. However, since this is caused by the characteristics of the I/O infrastructure and the runtime system of the Blue Gene/Q system, these read tests demonstrate also the scalability of SIONlib for this type of I/O operation.

The scratch file system of JUST has a maximum achievable bandwidth of about 160 GiB/s, whereas the aggregated I/O bandwidth of JUQUEEN is 370 GiB/s. This number can be computed from the results of the baseline measurements and the maximum number of 248 IONs.

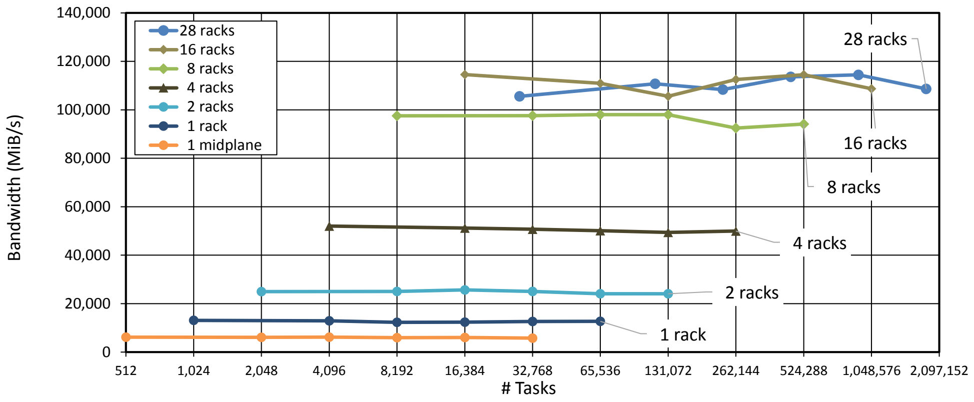


Figure 4.10: Measurement of writing data with SIONlib multi-file I/O on JUQUEEN up to the full scale of the machine (1-64 tasks per node, 256 MiB file data per node).

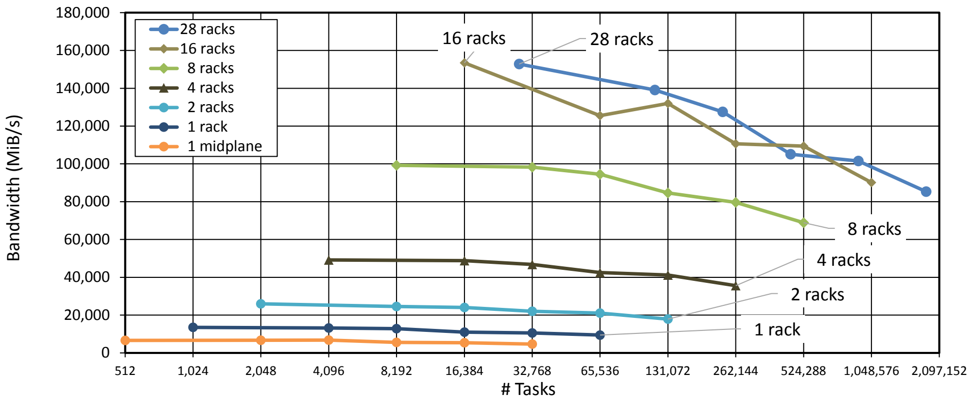


Figure 4.11: Measurement of reading data with SIONlib multi-file I/O on JUQUEEN up to the full scale of the machine (1-64 tasks per node, 256 MiB file data per node).

Taking both I/O bandwidths into account, the expected saturation point of the I/O write bandwidth should be at a scale of 12 racks, which is confirmed by the results shown in Figure 4.12. The achieved maximum I/O write bandwidth is at 114 GiB/s, which is 71 % of the maximum I/O bandwidth of the file system. The achieved I/O write bandwidth follows the maximum available bandwidth provided by the involved IONs, as long as the file system is not saturated. Therefore, SIONlib allows to achieve a significant percentage of the maximum available file-system bandwidth, even at largest scale on JUQUEEN.

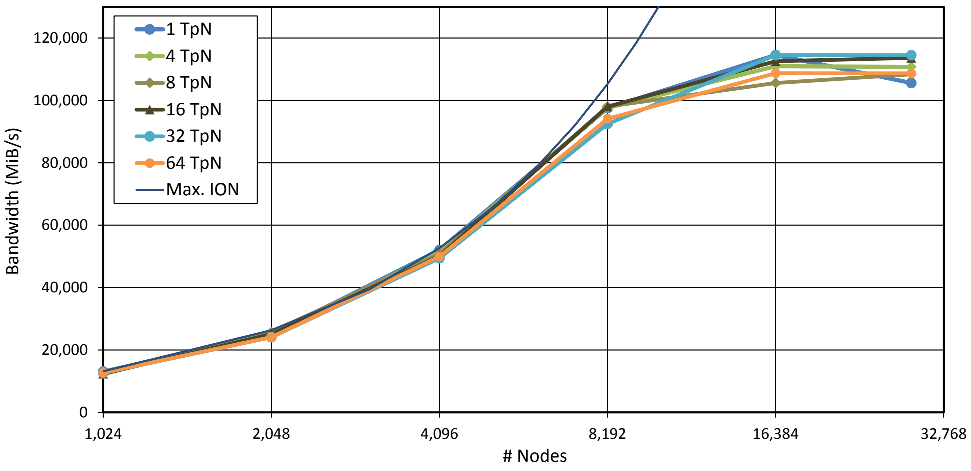
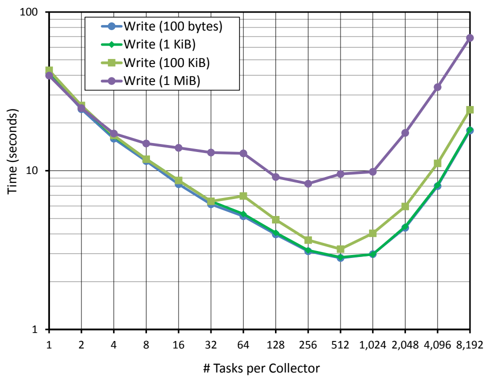


Figure 4.12: Comparison of write bandwidth on JUQUEEN for different number of tasks per node (TpN). 'Max. ION' shows the maximum I/O bandwidth, which can be achieved by that number of nodes (1-64 tasks per node, 256 MiB file data per node).

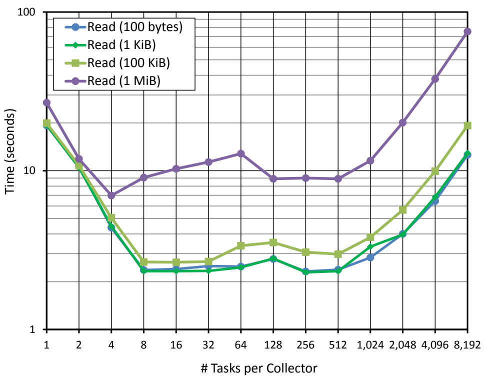
4.2.6 Small data I/O at large scale

With the coalescing approach, SIONlib supports applications that require to store only a small amount of data per task (cf. Section 3.8). Because SIONlib normally extends chunks to the minimum size of one file-system block, this would lead to a high amount of unused disk space in the SIONlib file container without using the coalescing feature. SIONlib solves this issue by aggregating data collectively on a smaller number of *collector tasks* that write the data to the file container on behalf of the other tasks. A critical configuration parameter of coalescing I/O is the number of *sender tasks*, which send their data to one collector. SIONlib has implemented a heuristics to find a default number of collectors according to the specified chunk sizes. Optionally, users can overwrite this default number of tasks per collector (`collsize`).

Figure 4.13 shows the results of a parameter study on JUQUEEN to find the optimal parameter ranges for different chunk sizes. The tests were performed on one midplane of JUQUEEN with 64 tasks per CN using one file per ION, which results in four physical files. As an example, the measurement of the write bandwidth with a chunk size of 1 MiB will be explained in the following. With a file system block size of 4 MiB and less than four tasks per collector, the file-system blocks are not filled in the SIONlib file container. Therefore, GPFS has to handle up to four times more file-system blocks. This leads to a linearly decreasing write time from one to four tasks per collector. In next task range from four to 64 tasks per collector, file-system blocks are completely filled (as all measurement points are multiples of four), which leads to nearly constant writing time. Up to a `collsize` of 64 tasks, one collector is located on each CN on average. This is changed with higher `collsize` numbers of 128 and more. In this case, collector tasks are running only on a subset of CNs and only those are actively communicating with the IONs. As I/O traffic is maintained on the ION with a multi-threaded daemon, the concurrency of the threads will be reduced with decreasing number of collectors, which has

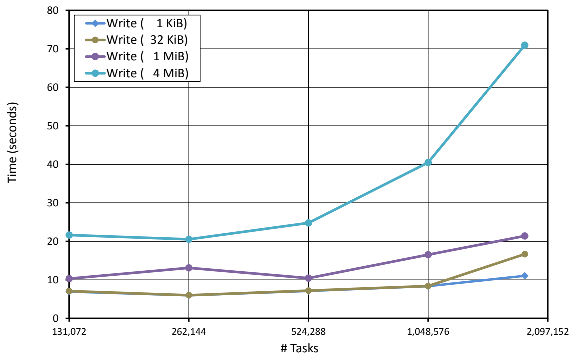


(a) Writing data

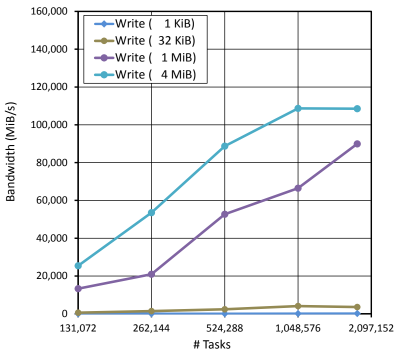


(b) Reading data

Figure 4.13: Evaluation of the influence of the parameter `collsize` on the time for writing and reading data of different size with coalescing I/O on one midplane of JUQUEEN (64 tasks per node).



(a) Write Time



(b) Write bandwidth

Figure 4.14: Scalability of coalescing I/O with SIONlib on JUQUEEN, using a collsize of 512 tasks: write time and bandwidth for different data sizes and numbers of nodes (64 tasks per node).

another positive effect on the writing time. The writing time decreases further until 256 tasks and remains constant up to 512 tasks. Starting from 1024 tasks per collector, the bandwidth decreases linearly from initially 3500 MiB/s to 500 MiB/s at 8192 tasks per collector. The reason for this decrease is that not enough I/O streams to each ION are in use. The bandwidth of one I/O stream seems to be saturated at less than 500 MiB/s. Consequently, at least four collectors should be used per ION to fulfill the overall ION bandwidth of about 2 GiB/s. The results of the read time and the runs with smaller data sizes show a similar behavior. However, the runs with smaller data size are faster because the collectors have to transfer less data.

Resulting from the discussion above, the selection of a `collsize` of 512 tasks is optimal for the tested data sizes. Therefore, this number of tasks per collector was configured for the scalability benchmarks that are shown in Figure 4.14. The benchmarks were run for data sizes from 1 KiB to 4 MiB from two racks of JUQUEEN up to the full system. Because of the reduced number of tasks that interact with the file system, the tests were configured to create one file per ION. While the two tests with the small chunk sizes of 1 KiB and 32 KiB created only small data sets of less than 56 GiB, the other two tests with 1 MiB and 4 MiB were dominated by the I/O bandwidth. For example, the latter test with 4 MiB chunk size created about 7 TiB of data on disk on 28 racks. The coalescing approach of SIONlib scales for the two smaller data sizes constantly up to one million tasks, which leads to a writing time from below ten seconds up to only 17 seconds at full scale with 1.8 million tasks.

As coalescing I/O is intended for large-scale applications with small data size per task, the measurements demonstrate that SIONlib can enable also task-local I/O for those applications without structural changes to the code. Additionally, Figure 4.14 shows that the coalescing approach also scales for applications with larger chunk sizes. The achieved write bandwidth of 108 GiB/s for a chunk size of 4 MiB is in the same range as the full-scale measurements without coalescing I/O (cf. previous section).

4.3 File System as a Shared Resource

The file system is a shared resource and multiple applications are using it at the same time, which makes it very likely that an application's I/O interferes with I/O operations of other applications. However, a single application cannot be optimized to reduce such an interference, because the necessary information about I/O activity of other application is not available. In a more detailed view, applications are typically using multiple I/O servers to store their data according to the file striping rules of the file systems. For example, GPFS implements a homogeneous file striping over all file servers and does not allow applications or users to influence the striping. On the other hand, the Lustre file system provides command line tools that allow users to configure the file striping for the files individually for each application. As a result, the distribution of the usage of Lustre I/O servers (OSTs) may not be homogeneous. Figure 4.15 shows an example for such a heterogeneous distribution of I/O workload. All parallel applications that perform synchronous parallel I/O to shared or individual files on the OSTs will suffer from such an uneven distribution of I/O load.

Figure 4.16 shows, as an example, the results of measurements on Jaguar, a Cray XT system at the Oak Ridge National Laboratory in the US, which has meanwhile been replaced by its successor Titan [75]. Jaguar was combined with one of the largest Lustre installations, which hosted the scratch file system for applications. The SIONlib benchmarks were performed in a configuration, where the physical files were assigned one-to-one to the available OSTs, which allows measuring the writing time individually for each OST. The individual timings are shown in the inner diagram of Figure 4.16. More intuitive is the plot in the outer diagram. For each time point (x axis) it shows the number of tasks that could finish the write operations in that time. The write operation of the first task ended after 6.2 seconds, whereas the last task required ten times longer to finish the write operation (62.9 seconds). Because an application's I/O routines are typically called synchronously by all tasks, the application cannot continue computation before the last task has finished the I/O operation. As the measurement indicates, a single slow OST can slow down the whole parallel application. To circumvent this, an application has to omit using slow OSTs. Desirable would be an adaptive mapping of OSTs to files, depending on the actual load on the OSTs. Since Lustre does not provide this kind of information, the application needs to test the actual I/O bandwidth of each OST itself. This technique has been proposed recently as an extension to parallel NetCDF [86], which skips OSTs with limited bandwidth within the I/O library. For parallel task-local I/O, SIONlib

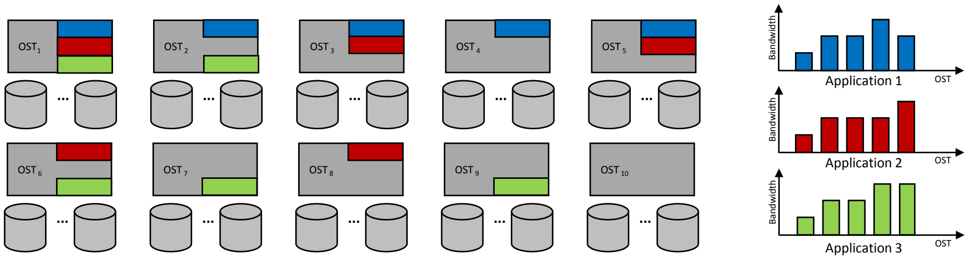


Figure 4.15: Example of a data distribution of three applications on Lustre OSTs (left) and the expected bandwidth per OST for applications (right).

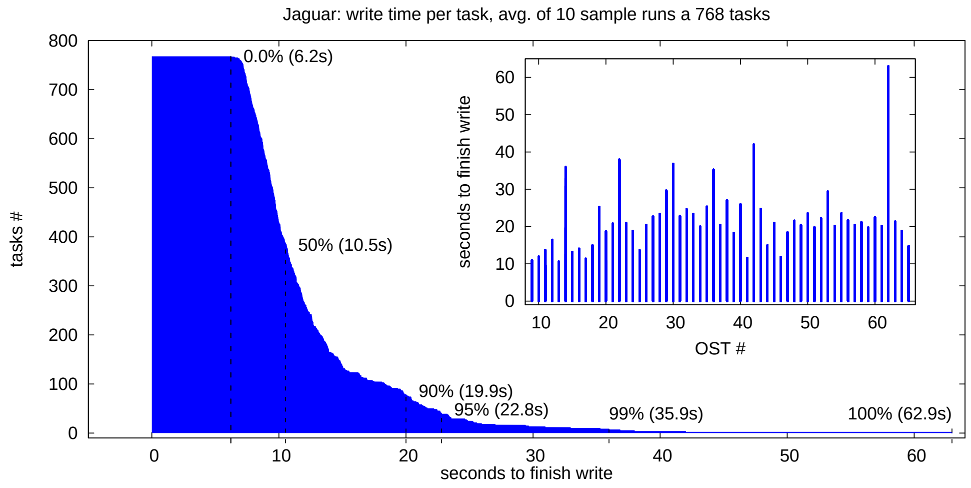


Figure 4.16: Write time per Lustre OST on Jaguar shown per OST (inner graph) and sorted by finishing time (outer graph).

provides with the multi-file approach a solution to map physical files and tasks to OSTs. This mapping scheme can be similarly extended by testing OSTs before using them and skipping OSTs with limited bandwidth. However, because the user-based configuration of Lustre striping does not allow specifying a set of OST numbers for a physical file, a physical file has to be mapped to one OST (cf. section 1.2.3). This is not a direct disadvantage, because this one-to-one mapping reduces the number of tasks accessing one physical file and therefore the metadata overhead. Additionally, the metadata handling will be localized to the OST storing this file.

In a broader perspective, the described problem of shared usage of the file system should be solved on the system level. For example, similar to the I/O node mapping in the I/O infrastructure of IBM Blue Gene/Q systems the OSTs can be mapped to parts of the compute nodes. Depending on the number of OSTs and compute nodes and the granularity of the distribution, this would dedicate OSTs to I/O streams of a single application, which is running on these compute nodes. One example for such an *I/O zoning* is the FEFS file system of the K computer at Riken [79]. The Lustre OSTs of the local FEFS scratch file system are physically and logically located adjacent to a compute-node partition. Data on the local file system can only be accessed by the local compute nodes, whereas the name space is globally managed by a Lustre MDS. SIONlib can directly support such distributed and separated file systems with its multi-file approach.

4.4 Applications

There are a number of applications that have already integrated SIONlib to improve parallel task-local I/O. Mostly, SIONlib is used to replace task-local POSIX I/O in checkpointing routines. Among these applications are the parallel tree code PEPC [39], muphi [92], a code for the simulation of water flow and solute transport in porous media, and the simulator for spiking neural network models NEST [81]. Another application using SIONlib is the Korringa-Kohn-Rostoker Green function code for quantum description of nano-materials (KKRnano) [49]. This code benefits extremely from the coalescing-I/O feature of SIONlib, as the checkpoints consist of task-local data with a size of not more than 100 KiB. SIONlib enables this application to run on the full JUQUEEN system while writing checkpoints in a reasonable time.

In the remaining section, two further examples for the integration of SIONlib are shown and corresponding measurements are discussed. The first one is the integration of SIONlib into the simulation code MP2C to support efficient checkpointing at large scale. The second example describes the integration of SIONlib into the parallel performance tool Scalasca, which has high I/O demands to store intermediate event trace files.

4.4.1 Checkpointing in 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 Fortran90 code MP2C couples multiple-particle collision dynamics, an established mesoscale simulation approach, with molecular dynamics. The current version of MP2C uses MPI and implements a domain decomposition approach, where geometrical domains of the same volume are distributed across the different processes [74]. 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.

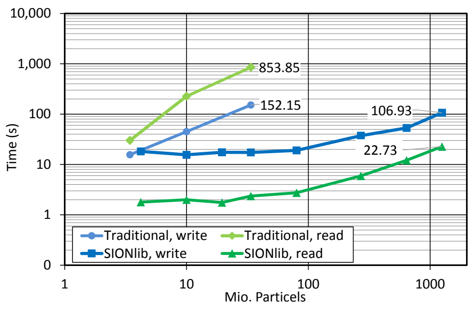
Traditional I/O approach

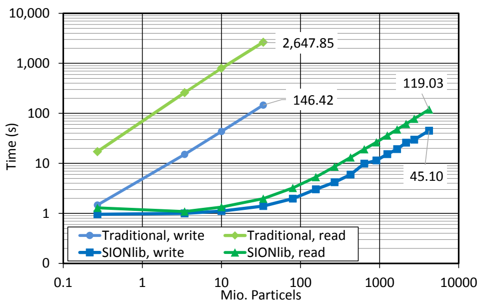
To avoid file handling issues that arise from having a potentially large number of files from the very beginning, the authors of the code had originally decided to follow the single-file sequential approach. In this approach, only one task of the application is selected to write or read data on behalf of the other tasks. Data will be transported in two steps to disk: first, the data is sent from the generating task to the writer task. Second, the data is written from the writer task to disk. In general, the data transfer bandwidth between compute nodes will be much higher than the bandwidth between compute node and disk storage. Therefore, only the I/O operations on the writer tasks and not the data transfer to the writer tasks are considered for further analysis. The performance of these I/O operations is limited by the speed of the single core on which the task is running and by the transfer speed to write data to the file system. A higher parallelization rate of the application cannot solve either limitation because only one instance writes the data and therefore uses only one data path to the file system.

Especially on a hierarchical I/O infrastructure, this serial approach will not scale very far. The essential I/O optimization on those architectures is to use parallel I/O over multiple data paths.

The individual data stream is limited. For example, on JUQUEEN one task is writing data over one I/O node, although the maximum write bandwidth on the I/O node is limited to 2 GiB/s (cf. Section 4.2.2).

First experiences on the Blue Gene/P system JUGENE, the predecessor of JUQUEEN, in 2009 showed that the scalability limitations of this approach resulting from serialized I/O in combination with alternating gather and write operations does not allow running MP2C effectively on more than 1,024 cores of JUGENE and with more than ten million particles. We could observe similar limitation on the current JUQUEEN system (cf. Figure 4.17). The author of the code has implemented the input of checkpoint data in a different way. Each task reads the whole data set into a temporary buffer and extracts the required data from this afterwards (*data sieving*). Apart from more flexibility, this approach reduces the read performance again. In the worst case, when no data caching is enabled, all tasks will read the data directly from the file system. This increases the overall read size by a factor equivalent to the number of tasks. The measurements on JUGENE and JUQUEEN confirm this (Figure 4.17) and show that on both systems the read time is an order of magnitude slower than the write time, although the Blue Gene I/O infrastructure provides read caching capabilities.





(a) JUGENE (1024 nodes, 1 task per node)

(b) JUQUEEN (512 nodes, 16 tasks per node)

Figure 4.17: Time needed by MP2C for writing and reading restart files at small scale on JUGENE and JUQUEEN with and without using SIONlib.

Integration of SIONlib

The particle data is distributed over all tasks of MP2C, because each task stores only those particles that are located inside a sub-box of the simulated 3D-volume that belongs to the corresponding task. Depending on the forces that are applied to the particles a movement of particles between sub-boxes and therefore between tasks is possible. To identify the particles over the whole simulation MP2C assigns a unique identifier to each particle, which is also stored in the checkpoint data. These characteristics of the application allows an easy implementation of parallel I/O to task-local files, where each task writes its data to a separate physical file. Since having each task writing its restart data to a separate file was no option due to the limited performance of file creation, MP2C is a suitable candidate for SIONlib. After modifying approximately 50 lines of code in such a way that each task writes 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 instead of 33 million particles with the traditional approach. [30, 89].

Figure 4.17a shows results from this first integration step of SIONlib on JUGENE. The graph compares the time needed by MP2C to write and read restart files on 1,024 cores of JUGENE with and without using SIONlib, respectively. The measurements were taken on a single rack in SMP mode. The 1,024 task-local files were mapped with SIONlib 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 the SIONlib approach amortizes only for larger problem sizes, where they are significant. For 33 million particles, the I/O performance was improved by 1-2 orders of magnitude.

Coalescing I/O

The integration of SIONlib and the automatic alignment of chunks to file-system blocks introduced the new optimization requirement to the application in order that task-local data should fill file-system blocks as much as possible to achieve good I/O bandwidth. The reason for this requirement is that chunks of different tasks are aligned to file system block boundaries and file systems typically have to handle and to write full file-system blocks to disk although they might be filled only partially (cf. Section 3.2.1). Configuration and application sizes of typical production runs of MP2C show that this optimization requirement is violated. The number of particles per task in such runs is only in the range of 10,000 to 30,000. MP2C stores 56 bytes in a checkpoint file for each particle. This leads to a maximum chunk size of 1.6 MiB, which is less than 50 % of the file system block size on the GPFS file system (4 MiB). Moreover, MP2C will potentially move particles between tasks during run time. This will change the chunk sizes from checkpoint to checkpoint, which will make an alignment optimization difficult.

Therefore, the integration of SIONlib was modified to use coalescing I/O. The usage of this feature avoids the disk space overhead and an I/O performance degradation caused by small data chunks (Section 3.8). A prerequisite for coalescing I/O is that the SIONlib calls have to be collective. MP2C fulfills this prerequisite, because checkpoint I/O is arranged in one subroutine and this routine is called collectively by all tasks at the same time. Figure 4.18 shows measurements of MP2C checkpoint operations using the SIONlib coalescing feature with different numbers of tasks per collector. Similar to the results of measurements with the SIONlib benchmark program (Section 4.2.6), the optimal number of tasks per collector is 512. Therefore, this value was used in all subsequent measurements with MP2C. In the current version of SIONlib, coalescing I/O operations are implemented using the two-phase I/O scheme, where a collector task collects the data from a number of tasks and writes it to the file system on behalf of these tasks. This scheme is executed for each collective I/O call separately, because SIONlib does not buffer the data internally. Consequently, a collector task can only write data consecutively to disk when all tasks write all data of their chunk in one I/O call. Otherwise, multiple write calls will force the collector task to write multiple times smaller data chunks at different positions in the file. However, such a striped file access pattern degrades the I/O bandwidth. To avoid such a bandwidth degradation, the checkpoint routine of MP2C was adapted to combine the I/O operations to one I/O call. Future versions of SIONlib will integrate the buffering and coalescing feature. Then, data will be buffered on sender tasks and data will be sent in one chunk to the collector task.

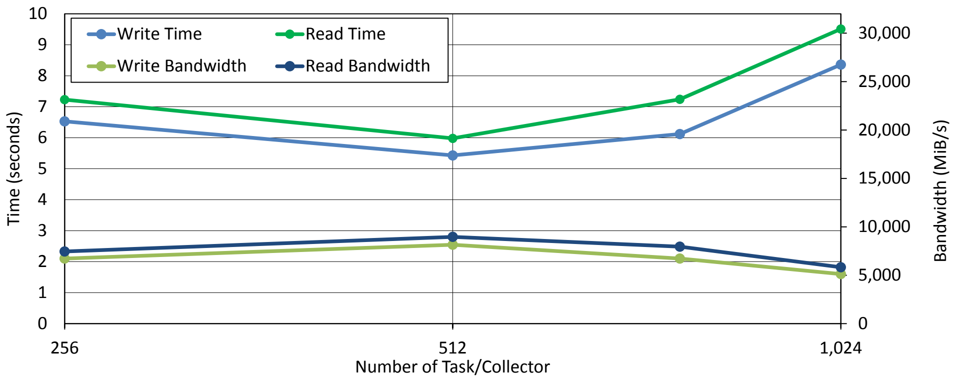
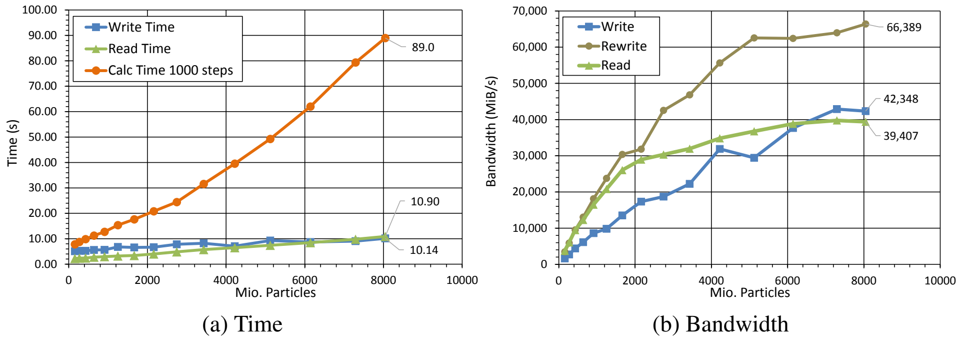


Figure 4.18: SIONlib coalescing tests with MP2C with different numbers of tasks per collector. Bandwidth and timings for writing and reading a checkpoint file are shown for one rack of JUQUEEN (64 tasks per node, 311 million particles).

Scaling to the full system of JUQUEEN

Beyond performance improvements for 1 K cores on JUGENE or 8 K cores on JUQUEEN, adopting the 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 4.19a shows the time needed for checkpoint I/O from 512 K tasks on JUQUEEN. The data chunks of the tasks were mapped onto 16 physical files per rack, which are 128 files in total. The measurements indicate that good performance can be achieved for a wide range of local data sizes using the coalescing I/O feature of SIONlib. The overall data size varies from 8.3 GiB at 156 million particles to 0.42 TiB when simulating 8 billion particles. In all test cases, SIONlib was able to manage writing the data from 512 K tasks to file in less than 11 seconds. For small data sizes, the latency of the I/O operations dominated the I/O time. For large data sizes, the overall available bandwidth to the file system was a limiting factor. The measurements show that this effect will start at 4 billion particles for write operations and at 2 billion particles for read operations. The earlier saturation point and the smaller read bandwidth compared with the write bandwidth are



(a) Time (b) Bandwidth

Figure 4.19: Bandwidth and time of MP2C checkpoint I/O for different numbers of particles on JUQUEEN using 512 K tasks on eight racks (64 tasks per node).

related to the I/O infrastructure of Blue Gene/Q, as explained in Section 4.2.2.

MP2C only needs the latest checkpoint file after the successful completion of a simulation job for a restart. Therefore, former checkpoint files can be overwritten. The measurements show a higher bandwidth for overwriting, which could be explained by less overhead in the file system: in this case, the file system does not have to create the files and to allocate file-system blocks because they are already allocated and assigned to the files. Figure 4.19b indicates a bandwidth improvement of nearly 50 % when overwriting an existing checkpoint file in MP2C. In this case, the maximum time for writing a checkpoint will be reduced to 6 seconds. The checkpoint frequency therefore only depends on the calculation time, which increases with the number of particles.

With SIONlib, the scaling tests could be pursued up to the full system size of 28 racks, running MP2C with 1.8 million tasks. The total number of particles could be increased to 270 billion particles. Figure 4.20 shows the achieved bandwidth for different particle numbers. At largest scale, the file-system bandwidth measured with the SIONlib benchmarks could be confirmed (cf. Section 4.2.5). The checkpoint file had a size of 14.4 TiB at this scale and was written in 147 seconds (read in 108 seconds). Since the file-system bandwidth starts to be saturated at about 20 billion particles, the time for checkpointing remains constant up to this number of particles and increases from there linearly.

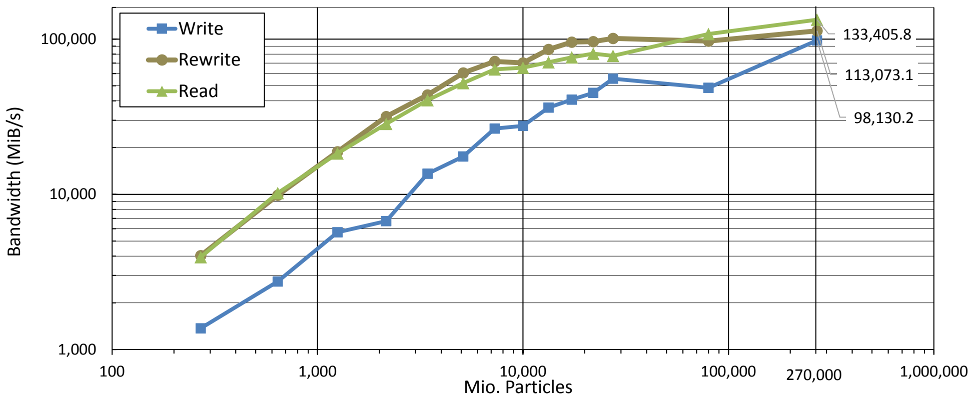


Figure 4.20: Achieved bandwidth running MP2C with SIONlib support on full JUQUEEN system for different particle numbers (28 racks, 1,8 million tasks, 64 tasks per node).

Integration into the application workflow

Writing data to task-local files or to SIONlib containers has implications for the subsequent data processing steps, unless the data is read from an application running with the same number of tasks. Only in this case, individual data files or chunks of a SIONlib file can be assigned to reader tasks in a one-to-one fashion. However, typical post-processing steps will not run on such large number of tasks that are needed for the simulation runs. In those cases, the post-processing tool will run in an *m:n*-scheme, where *n* is the number of data files or SIONlib chunks and *m* the number of consumer tasks in the post-processing program. For serial data processing steps, the mapping scheme will degrade to *1:n*.

In contrast to high-level parallel I/O libraries, SIONlib only provides an interface for binary data and does not require to describe the data format and its structure. Typically, data from different chunks cannot be merged automatically without this knowledge about data format and structure. This is only possible in exceptional situations where data chunks can be concatenated without modification. A serial customized conversion tool for such a concatenation can be implemented with the serial API of SIONlib. Furthermore, a parallel conversion tool can be implemented with support of the *mapped open* feature of SIONlib, which allows SIONlib to open a file in an *m:n*-scheme (cf. Section 3.10.3).

At start time MP2C requires the sorting of particles according to their position in the 3D-simulation box. This requirement is fulfilled when the application restarts from a checkpoint file with the same number of tasks and the same problem configuration. In other cases, the particles have to be resorted according to the new configuration. In the traditional I/O approach of MP2C, this sort operation was done implicitly during reading, where each task reads all particles from the input file and selects those particles that are inside the local box. Although this scheme is practicable for small particle numbers, it is not usable at large scale or with large numbers of particles, as demonstrated in initial measurements of MP2C (cf. Figure 4.17 on page 89).

MP2C has a serial conversion tool to convert output data between the traditional format and SIONlib files. Typically, this tool is designed to convert data for post-processing, like visualization. For small data sets, this tool can also be used to resort the particles in two steps. First, the latest checkpoint file in SIONlib format will be converted to a single data file in traditional format. Second, at start time of MP2C the input routines for the traditional format will read the file and extract the local data (*data sieving*). Such a conversion is only needed in rare cases, where checkpoint data will be written and read from applications with different number of tasks. Typically, such a conversion is only needed at the initialization of a production run with multiple jobs in a job chain. SIONlib provides a much better I/O bandwidth for writing and reading intermediate checkpoint and restart files, so that the higher effort for such a data conversion is justified.

For larger use cases, a more scalable and flexible integration of the SIONlib format into the workflow of MP2C has to be implemented. The input routines of MP2C have to be modified, so that they can read checkpoint files with different number of chunks using the *mapped open* feature for data input with SIONlib. Depending on the number of SIONlib chunks and the number of consumer tasks in the post-processing program, tasks will read no data, one chunk, or multiple chunks from the input file. In general, the particle data will not map to the local box of this task. Therefore, non-local particles have to be sent to other tasks. Such a redistribution can be implemented in different ways. For example using the `MPI_Alltoall` collective communication function or sending data in multiple steps along a ring-topology from task to task with `MPI_Send`. In the latter method, all particle data will pass each task, and local particles can be fetched from the circulating data, similar to the traditional input scheme. For post-processing, either using the serial conversion tool or adding directly a SIONlib reader to the post-processing tool is proposed.

4.4.2 Trace-file generation with Scalasca

SIONlib has been integrated into Scalasca version 1.x to improve parallel task-local I/O of event traces from MPI and hybrid programs (MPI+OpenMP) at large scale. In the first step, trace files containing the event records are written from an instrumented parallel application to a SIONlib file container. In a second step, the trace files will be read again from the parallel trace analyzer. As an example, the results of measurements of I/O operations in the first step are discussed in this section. These measurements were performed on JUQUEEN with one of the sample programs of Scalasca (*ctest*). The results of the first test with the pure MPI variant of the program are shown in Figure 4.21. The runs were scaled from one midplane of JUQUEEN to two racks and were repeated for 16, 32, and 64 tasks per compute node. The sample program was configured to generate trace data with a size of 3.5 MB per task. The data is compressed during the write operation on-the-fly with a typical compression rate of 50 %. This results for the largest configuration in an output file size of about 200 GiB.

The trace-file generation consists of two different parts. The first part includes creating and opening the physical files, which is performed at program start before executing the first instruction. The second part, writing the event traces and closing the files, is done at the end of the program. Intermediate flushes did not occur in these tests, as the memory buffers of Scalasca were defined large enough to store all local event data. The I/O timings shown in Figure 4.21 are the summation of the timings of both parts. The I/O timings show that Scalasca with SIONlib support is faster than the variant using traditional parallel task-local I/O in each test case. Because the size of the task-local event traces is constant, the overall size of the generated trace files grows with the number of tasks. Furthermore, the overall I/O bandwidth is limited by the number of involved I/O nodes. This causes that the I/O time increases linearly with the size of the output files and consequently with the number of involved tasks.

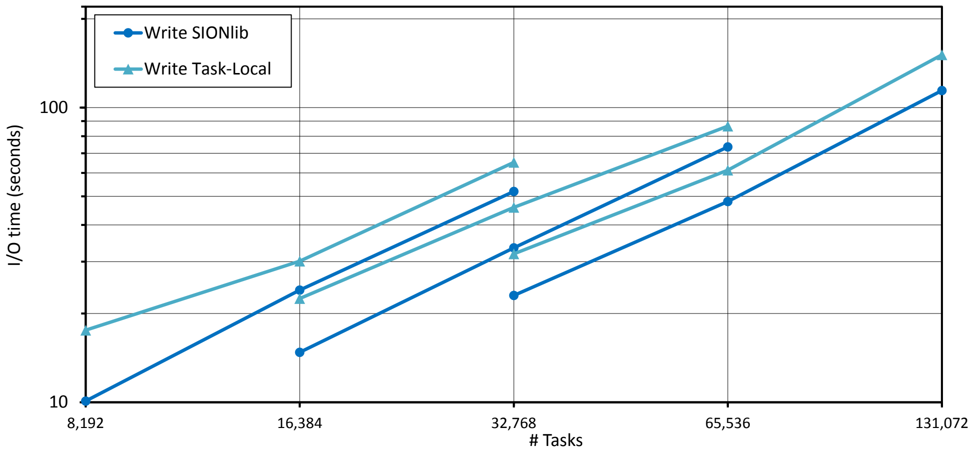


Figure 4.21: I/O timings for Scalasca's trace-file generation using traditional task-local I/O or using a SIONlib container with one file per I/O-bridge node. The tests were repeated for different number of tasks per node on one midplane, on one rack, and on two racks of JUQUEEN (3 curves, 16-64 tasks per node).

The subsequent read operation in the second analysis step of Scalasca could not be improved with SIONlib for pure MPI codes since the metadata handling of existing individual files does affect the read performance (cf. baseline measurements in Section 4.2.2).

In an additional test, the benchmark program was modified to run in hybrid mode by executing the inner loops within an OpenMP parallel region. In this case, Scalasca records for each thread of the parallel region a local event trace. Without SIONlib support, these event traces are merged on the thread level to one trace file per MPI task at the end of the program execution. Furthermore, each OpenMP thread reads the whole trace file of the corresponding MPI task and extracts its own data from the merged event trace. Scalasca with SIONlib support is not required to merge event traces on the thread level, because SIONlib provides space for each thread in the container file that this thread can access directly. Tests at small scale show that merging is very time consuming. For example, on 128 compute nodes, running `ctest` with one MPI task and 64 threads per compute node yields an I/O time of 391 seconds for the traditional approach and 25 seconds for the SIONlib approach, which is about 15 times faster. Further tests at larger scale with the traditional approach were not possible due to the overhead caused by the additional merging of trace data. Therefore, the evaluation of hybrid applications with Scalasca at larger scale can only be performed with support of SIONlib.

The on-the-fly compression of trace data in the measurement phase reduces the size of local data during writing. However, as SIONlib requires to know the chunk size in advance, this leads to unused space in the file container. Furthermore, as SIONlib reserves at least one file-system block per chunk, such small data will fill the chunks only partly. For example, in the previous measurement (Figure 4.21), the chunks are only filled half (less than 2 MiB/task). To improve Scalasca's I/O also for such configurations, the integration of SIONlib in the Scalasca measurement environment was prototypically modified in such a way that SIONlib with coalescing I/O operation could be used. This was achieved by applying an in-memory com-

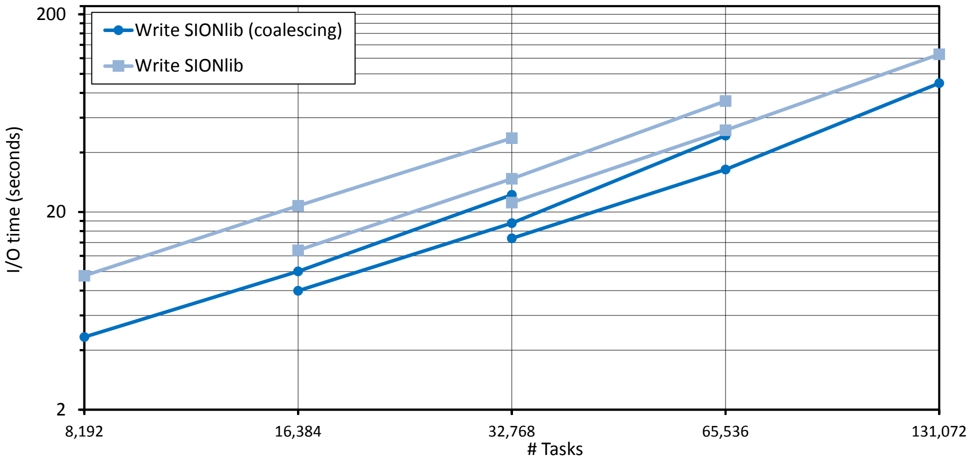


Figure 4.22: I/O timings for Scalasca's trace-file generation using standard SIONlib support and using SIONlib support with coalescing I/O. The tests were repeated for for different number of tasks per node on one midplane, on one rack, and on two racks of JUQUEEN (3 curves, 16-64 tasks per node).

pression of the event data at end of the program and calling the `reinit` function of SIONlib to reduce the chunk size according to the size of the compressed data afterwards (cf. Section 3.10). This makes the SIONlib file container denser and reduces the data transfer to a minimum. Figure 4.22 shows the results for a configuration with only 160 KiB compressed data per task. In comparison, the new approach has improved the I/O time in this measurement by 40 % on average.

Further tests of Scalasca version 1 with SIONlib support are performed on the *Stampede* system at Texas Advanced Computing Center (TACC) [94]. Compute nodes in this system are equipped by a Xeon Phi coprocessor. Depending on the usage model, application processes can run on both (*symmetric mode*), the host processor or the coprocessor, while MPI tasks on the coprocessor typically spawn a number of OpenMP threads. The tests in this study were performed on four *Stampede* nodes, running two MPI tasks on the host processor and 15 further MPI tasks on the coprocessor, each which 16 OpenMP threads (1088 threads in total). The study has demonstrated that SIONlib can support also the symmetric mode without modification by creating one physical file per MPI task.

5 Spindle

As demonstrated in Section 2.3 loading dynamically linked executables has limited scalability. The reasons for the increasing load time are the parallel, not coordinated look-up and read operations during load time. The dynamic loader, the software component that performs these operations, has been designed as a serial procedure, which itself refers to the serial POSIX I/O interface to interact with the file system. Therefore, the dynamic loader does not use application parallelism to coordinate look-up and read operations. In this chapter, Spindle is introduced, which offers new concepts to circumvent these limitations of the traditional dynamic loader. The main idea is to transfer parallel characteristics of the application to the layer of the dynamic loader.

The first section illustrates the objectives and the overall architecture of Spindle. The following sections describe the particular components of Spindle and its interaction with the application and the runtime system. The last section gives an overview of related approaches to this problem and highlights the unique properties of Spindle.

5.1 Objectives and Overall Architecture

Parallel applications that follow the SPMD model issue redundant load requests of DSOs across a potentially large number of processes (cf. Section 1.4.4). Most load requests occur at program start-up and are concentrated to a very short time interval. This is one of the issues that causes the type of file-access storm that was described previously.

An infrastructure that collects the I/O requests from the individual dynamic loaders and forwards only unique requests can avoid the file-access storm. This infrastructure is implemented in Spindle, which intercepts on the one hand requests for libraries made by the dynamic loader. On the other hand, Spindle handles the requests with a combination of scalable parallel broadcasts and local file system caches. The caches provide efficient local access, because the broadcast operations lead to a single load operation from the file system for each requested object. Loaded objects are then propagated to the nodes that need them. This removes the $O(P)$ scaling bottleneck, which was discussed in Section 2.3.1.

Additionally, with high probability, each process loads the same sequence of libraries at start-up, because link dependencies are embedded in the executable and libraries. This is especially given for SPMD programs. However, on large-scale systems, even applications that employ the MPMD model tend to divide processes into SPMD groups (cf. Section 1.4.4). For each of the SPMD groups similar sets of DSOs and load orders are given. The second objective of Spindle is to exploit this characteristic. Therefore, the Spindle broadcast operation can be initiated in two different ways. With the *push* model, Spindle assumes that when it receives

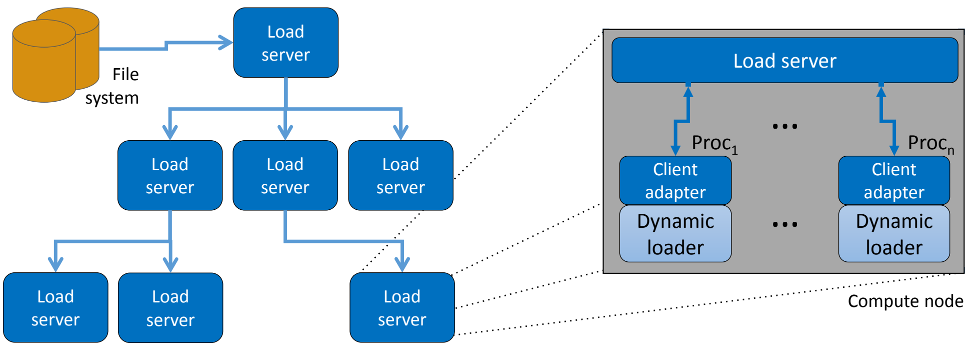


Figure 5.1: Overall architecture of Spindle. The client adapter is attached to each application process. The adapter communicates with the local load server, which interacts itself with other load servers via an overlay-network.

the first request for a particular library, other processes are likely to make the same request. With push, Spindle immediately broadcasts each library to all processes when it is requested the first time. In contrast, the *pull* model sends libraries to nodes only as they request them.

Figure 5.1 shows the overall architecture of Spindle for coordinating the loading procedure among different processes of a parallel application. It maintains a collection of load servers (daemons) alongside the application processes, which cache replicas of DSOs in local storage (e.g., RAM disks). A client adapter that is dynamically linked into each application process redirects all load requests to a nearby load server. In this way, the client adapter is kept as light-weight as possible. The servers are attached to a network that connects them to each other and to the underlying file system. With this network, load requests for an object, issued by different application processes, can be combined and data broadcasting can be efficiently facilitated.

As mentioned, application processes usually load most of their required libraries during start-up. However, MPI and other runtime communication systems are typically unavailable during this time. This makes it difficult to coordinate loading in parallel. This restriction leads to the first design decision: Spindle establishes its own independent communication network, implemented as an overlay network on top of the physical one. The load servers are designed to communicate with an arbitrary number of application processes and other load servers at the same time. This provides the flexibility needed to build customized overlay network topologies that optimally match the underlying hardware communication structure and the capabilities of the file systems. Figure 5.1 shows a tree topology with one root server, which is connected to the file system. In the future, it is planned that Spindle is extended in order to support additional topologies to be more efficient, such as a forest of trees whereby multiple roots connect to a more capable file system. Furthermore, the daemon concept keeps the design open for integration with other daemon-based system services. Therefore, it forms a precursor of a more general parallel loading service architecture.

Finally, the application client adapter has to reroute all file-system requests such as searching for libraries or loading the library code to the Spindle servers. To avoid implementing an own dynamic loader, the *rtld-audit* auditing API of the GNU loader is used for this purpose. This interface allows user code to intercept load requests and modify the dynamic loader's behavior. The design of Spindle allows all components to run in user space and without having to modify the runtime system.

The next sections explain the components of Spindle, including the client adapter, the load server, and the overlay network.

5.2 Client Adapter

Spindle has to interact with the dynamic loader. The responsible component of Spindle is the client adapter, which hooks into the load process of the dynamic loader by intercepting load requests and file searches. To implement the interception, the *rtld-audit* [78] interface is used, which is provided by the GNU dynamic loader. The interface allows users to register callbacks that are invoked when the dynamic loader performs certain operations. One of these callbacks (`la_objsearch`) is triggered before the loader tests a directory for the existence of a library. This callback can optionally return an alternate location that the loader should look into. The Spindle implementation of this function reroutes load requests to the Spindle load server, which obtains the DSO and returns its location in the local RAM disk. The callback routine then returns this location to the loader for access from the faster local storage on a RAM disk (cf. Figure 5.2).

The *rtld-audit* interface was originally intended to help debug the dynamic loader, thus enabling it has several implications when used for Spindle. Most of the callback functions implement a read-only access to loader-internal data. One exception is the `la_objsearch`, which is used by the Spindle client adapter to modify the load path of a DSO. There is, for example, no callback function to influence the library read operation itself, because data will be read directly via POSIX calls from the file system. It is not possible to pass library data, which is already stored in memory, directly to the loader. Therefore, Spindle first stores the data on a local RAM disk and passes then only the new location to the loader.

The callbacks for *rtld-audit* are supplied in a DSO, which is specified via the `LD_AUDIT` environment variable, and loaded into a special library namespace. This namespace protects the *rtld-audit* library by disallowing other application libraries from seeing its symbols. The

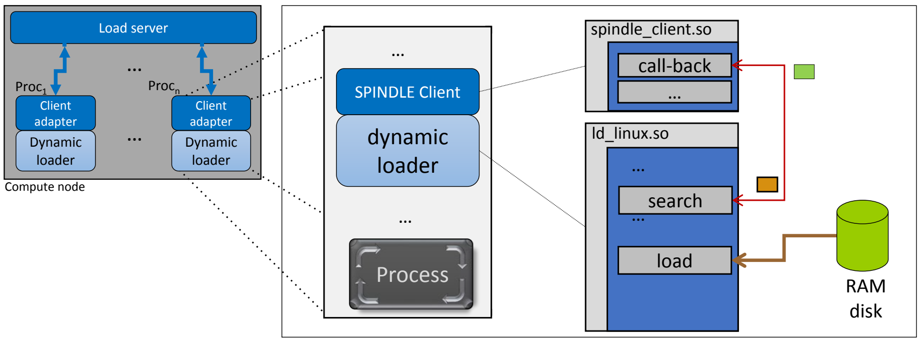


Figure 5.2: Spindle client adapter. The client adapter implements a number of callback functions, which are called from the dynamic loader at runtime.

strong isolation also restricts the access to functionalities in other libraries, with an exception for *libc*. Thus, the client adapter cannot use the TCP/IP stack, which resides in additional dynamic libraries. Instead, Spindle uses Unix-named pipes to communicate with the load servers, which can be managed directly using *libc*. Because named pipes do not allow inter-node communication, Spindle requires currently that at least one load server is running on each node.

Furthermore, enabling *rtld-audit* puts the dynamic loader into debug mode and has a performance impact on the application. Under normal execution mode, the first time an inter-library call site is invoked, the dynamic loader will look up its target function and cache it in the global offset table (GOT, cf. Section 1.4.3). Subsequent invocations of the call site will use the GOT entry and skip the look-up. This binding has performance advantages, in particular, when this call site is executed multiple times. With *rtld-audit* enabled, however, the loader will not modify the GOT entry, which is needed to ensure that *rtld-audit* callbacks are invoked for each inter-library call. Without caching the look-up results in the GOT table, the debug mode of the dynamic loader imposes an additional performance overhead, which is addressed by having the client adapter take over the responsibility of filling the GOT for the dynamic loader.

The *rtld-audit* interface does not provide any mechanism for intercepting load requests for the executable or the *rtld-audit* library. However, loading the executable through Spindle is particularly important because executables can contain a large percentage of application code and cause significant load on shared file systems. Spindle solves this using LaunchMON [1], a scalable tool infrastructure for launching parallel applications and tools across a wide range of HPC platforms. The user gives Spindle the command that launches the parallel application. Spindle then modifies the command to launch a small statically linked bootstrapper executable first and transfers control to LaunchMON. Next, LaunchMON starts this bootstrapper alongside the load server on each node. Finally, the bootstrapper works with the load server to move the executable and *rtld-audit* library to the RAM disk and then executes the application using the `exec`-system call.

5.3 Load Server

The second component of Spindle is the load server. As illustrated in Figure 5.1, the load server act as independent daemon process, which run alongside the application processes on each node. As described in the previous sections, the load server has to run on each compute node due to the limited communication functionality of the client adapter. The load server communicates on the one hand with the Spindle client. On the other hand, it communicates with the other load servers running on other compute nodes. This configuration allows the daemon to use local storage (e.g., local disks or RAM disk) to store library data and to allocate memory to manage internal data structures.

The daemon will consume most of the CPU cycles during the start-up of the application. During application execution, the daemon will sleep, while it is waiting for client requests. Thus, the Spindle daemon activity will not disturb the application. Besides CPU cycles, memory is another resource that has to be shared with the application. We will discuss the memory usage of the Spindle server in Section 5.3.2 in detail and will show how to reduce the memory overhead.

5.3.1 Operation

To handle DSO search and load requests for clients, the load server has three roles: it manages local replicas, interacts with other load servers through an overlay network, and accesses the file system if it is designated to perform the initial read of the library data. Locally, the server stores copies of DSOs and metadata of search directories and libraries. The files are stored in a RAM disk, and the server maintains the metadata in a memory-based data structure.

Two different schemes of data management in the Spindle server can be distinguished: The first scheme is metadata handling, where the test on file existence contributes a significant part to the load on the shared file system. Spindle optimize these operations by caching the contents of directories (at the time of the first look-up) instead of single file metadata. When the dynamic loader tests a file path for existence, the operation is rerouted to the load server. The server looks up the corresponding directory in its metadata table. If it exists, the table provides information of all files in the directory and the existence of a file can be verified locally. Otherwise, the load server triggers a designated-reader protocol, whereby a server responsible for the directory reads it via file-system operations and broadcasts the results to the rest of the servers. Figure 5.3 illustrates this metadata handling: on a search request, a server either parses the directory by itself or forwards the search request to another nearby server in the overlay network. The metadata table is implemented in Spindle as a hash-table, which allows fast search operations.

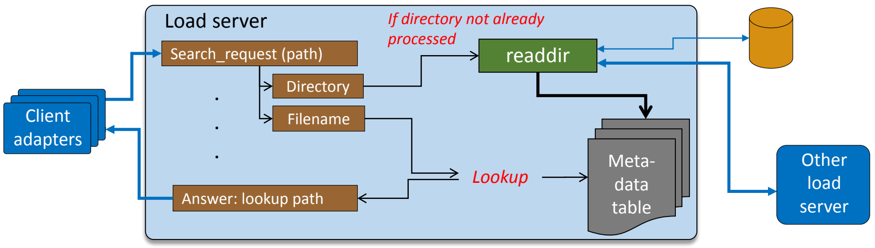


Figure 5.3: Handling of look-up requests by the Spindle server. Metadata will be processed for whole directories. If an object in a certain directory is searched, the metadata (e.g., file names) of the directory will be stored in the internal data structure.

The second scheme is the handling of the DSOs itself: One server reads the file from the file system and distributes it to the remaining servers, which store it on their local RAM disks (cf. Figure 5.4). In terms of data management, Spindle can apply a *replication* scheme, where all objects are pro-actively replicated across servers, as well as *distribution* schemes, where data distributed among the servers reactively services requests. The first scheme is supported by Spindle's push model; the second scheme is served by the pull model of Spindle. Although replication requires more memory, it offers a significant performance advantage over the alternative. The load requests typically occur in the same order in a short interval, and thus the replication-based push model can better exploit such strong locality of reference. For example, the replicated metadata allow the bulk of search operations to be satisfied locally without relying on frequent server-to-server communication. Library files themselves will also be locally available, shortly after one server reads the file from the file system, ready to service other identical requests expected in near future.

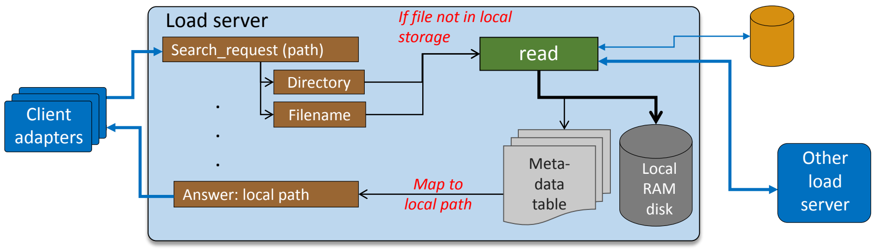


Figure 5.4: Handling of load requests by the Spindle server. One server reads the library file from the file system and distributes the data to all other servers. The local storage is typically a RAM disk.

Figure 5.5 shows the implementation details how the load server reacts on the different requests from different input sources. The Spindle clients are connected with named pipes to the local server. Requests from local clients will occur in a non-deterministic order, because the dynamic loaders of different processes are not coordinated. Therefore, the server has to listen on all incoming pipes to respond to client requests. A similar situation is also given on the server side: In the tree topology, the server is connected to a number of child servers and to a parent server if the server is not the root node of the tree. Requests can arrive from both sides. The parent sends for example metadata and file data to its children. Child nodes send look-up and file requests to its parent nodes. Additionally, requests for new connections can occur during the initialization phase. Such additional requests can originate from new client processes that are forked from existing processes. Therefore, the server has to listen also for new connections. In the case of name pipes, this has to be implemented by monitoring the directory, in which new pipe files will be created.

All sources the server has to monitor are or can be mapped to POSIX file descriptors. With the POSIX `select` function, all sources can be monitored at once. The Spindle server will run this call in a loop and responds each time one of the sources is becoming active. The server provides a special request handler routine for each type of request and source, which performs the corresponding action and submits new requests or acknowledgments to other connections if necessary.

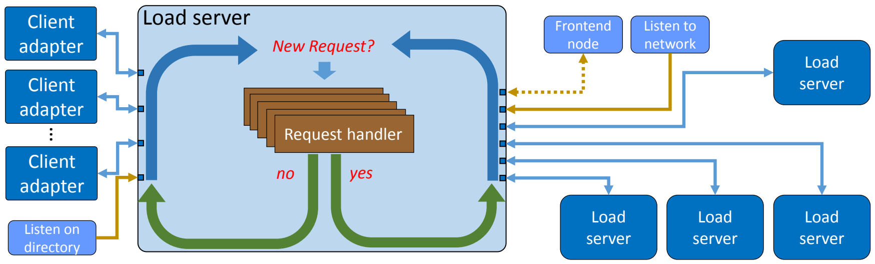


Figure 5.5: Scheduling of requests by the Spindle server. The Spindle server is listening on all connections, and responds to incoming requests by selecting one of the request handlers. As a result, the request handler may send further requests to other components of Spindle.

5.3.2 Memory overhead

As described in the previous section, Spindle uses a part of the memory, the RAM disk, to store library files. We discuss in this section the implications on applications when library file data is handled in such a way. The virtual address space of an application process on Linux systems is divided into memory pages. These pages can be loaded separately from disk into memory, which happens typically at *page fault* events. Such an event is issued when the processor executes instructions or access data that is located in a page existing only on disk. The processor has to stop execution of the code and to wait until the page is loaded into memory. Depending on memory space, pages can also be swapped out, for example, when space is needed and the page was not used recently. The pages that are used during runtime and that are stored in memory are described as the *working set* of an application process. Figure 5.6 (upper part) depicts a situation where part of the pages are loaded into the memory (working set) and the rest of the pages resides only on the file system. On HPC systems, the files are typically stored on a remote file system. Therefore, each page fault will issue a request to the remote file system to read the data if the file is not already stored in a local file system cache. This page-load strategy is not only used for the program executable, it is also applied to dynamically linked library files.

Spindle loads the whole library file to a local RAM disk. The code is stored and kept in physical memory for the entire execution of an application. Fortunately, the RAM disk is not a direct duplication of code pages; Linux is smart enough to use the same physical memory to back both the RAM disk and the application's virtual memory pages. Thus, pages of the working set are only stored once in memory. However, also the unused pages are stored in the RAM disk. The bulk of Spindle's memory overhead thus comes from placing code pages into the RAM disk that would not have been touched or loaded without Spindle. Figure 5.6 (lower part) illustrates this situation. The memory overhead is approximately equal to the total amount of code in libraries and the executable minus the amount of code in the application's working set. In other words, Spindle makes an application use memory as if every code page was in its working set.

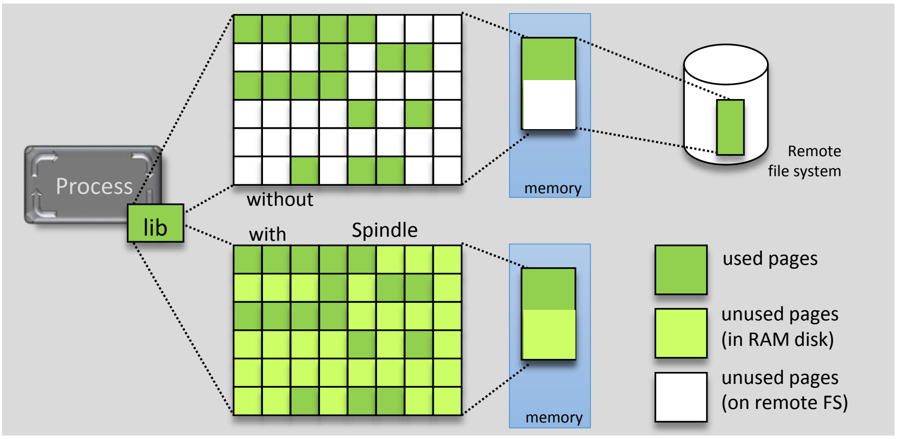


Figure 5.6: Memory usage of an application process without and with Spindle support.

Furthermore, memory used by libraries that are loaded by Spindle is treated differently in low-memory situations. Since the memory pages that store libraries are backed by a RAM disk, they cannot be paged out when the system runs low on memory. If those pages were backed by a shared file system, they could be dropped and reloaded as needed. This means that an application using Spindle may run earlier out of memory as if it does not use Spindle.

While code preloading causes extra memory overhead, it has advantages for extreme-scale environments. Loading pages from the file system as needed during execution is a well-known source of undesirable OS noise. Thus, high-end systems like IBM Blue Gene intentionally pre-load pages into memory and avoid this noise [38]. On such systems, Spindle would not use extra memory. Even on other systems without preloading of pages, Spindle would avoid undesirable OS noise, which caused by on-demand loading of pages.

As an optimization, Spindle does not load the entire library file into the RAM disk. Each DSO specifies parts of itself that should be loaded into memory via a table of *program headers*. Typically, a DSO will specify that its code and data should be mapped into memory, while debug information and the symbol table are left on disk. The load server reads these program headers and caches only the parts of a library that are needed at runtime. Essentially, Spindle strips libraries before transmitting them. This does make using a debugger on a Spindle application difficult, though it is planned to hide these effects by pointing debuggers back to the original library files.

To conclude the discussion about memory overhead, the concepts of Spindle improve the performance and scalability of shared library loading by the trade-off that Spindle requires extra memory on each node.

5.4 Overlay Network

The dynamic loader becomes active very early in the startup phase of application processes. The client adapter intercepts the dynamic loader and redirects load requests to local copies of library files. A network of load server is established, which take over this task. To funnel requests to the file system and to broadcast DSOs back to the application, Spindle load servers have to rely on a communication infrastructure. However, the load servers are running as separate daemon processes outside the application domain, which makes it difficult to use the infrastructure of this domain for Spindle communication purposes. Additionally, at start-up time, when the dynamic loader is active, the communication infrastructure of the application (e.g., MPI) has not been initialized. This is because shared libraries that contain the communication functions have not been loaded. Furthermore, the application does not have execution control at this time and initialization calls (e.g., `MPI_Init`) have not been executed. Therefore, an own communication infrastructure has to be established, an overlay network on top of existing communication software stacks.

To avoid at large scale communication bottlenecks in the way how Spindle distributes libraries, an efficient network topology has to be used. Typically, this is a tree topology, which reduces the number of communication hops for broadcast operation to a logarithmic scale. Spindle uses for this COBO [19], the Collective Bootstrapper, which is a scalable implementation of a collective protocol named *PMGR*. This protocol is used as an MPI job start-up mechanism. Unlike the original PMGR protocol, which connects all clients to one master client, COBO uses a scalable tree topology. COBO is part of the LaunchMON software infrastructure [1], which allows tool servers to be co-located with a parallel job. Spindle integrates LaunchMON to start the load servers along with the application processes. The overlay network is initialized during application start-up. Given a list of hosts as input, COBO establishes the overlay network by distributing connection information from the root down to the leaf nodes. Given that the servers — in contrast to the clients — are not part of the parallel application and are not restricted to *libc* functions, the TCP/IP sockets can be used for inter-server communications. With COBO collective communication operations, metadata and library data are distributed from the root Spindle server in a binomial tree topology to the load servers of Spindle. To connect n nodes, the resulting binomial tree has a order d ($n \leq 2^d$). According to the properties of such a tree, messages broadcast from the root will arrive at each node after at most d hops. Thus, tree depth and walk distance vary no more than logarithmically with the number of load servers. Figure 5.7 illustrates this data distribution scheme for a tree with order $d = 3$. Nodes that receive a data packet from their parent nodes first forward the packet to the child nodes, starting with the outermost one on the left. Afterwards the nodes process the data. Figure 5.7 shows the data distribution for a binomial tree with eight nodes in which all nodes have received the packet after three steps. The data processing will start therefore with this distribution scheme on all nodes at the same time.

The main communication direction in Spindle is broadcasting data from the root node to the inner and leaf nodes. However, for applications requiring the pull model of Spindle (e.g. MPMD), requests have also to be transmitted back to the node that has to interact with the file system. Therefore, COBO has been extended by adding functions that provide point-to-point messages targeting the root node of the tree. With this modification, Spindle realizes with the tree topology both models push and pull for request handling.

The presented implementation of communication with a tree topology is adequate for configurations where the data source is at a single node. In Section 5.7, we will discuss a more general approach, which allows flexible topologies and multiple nodes that are responsible for file-system interaction.

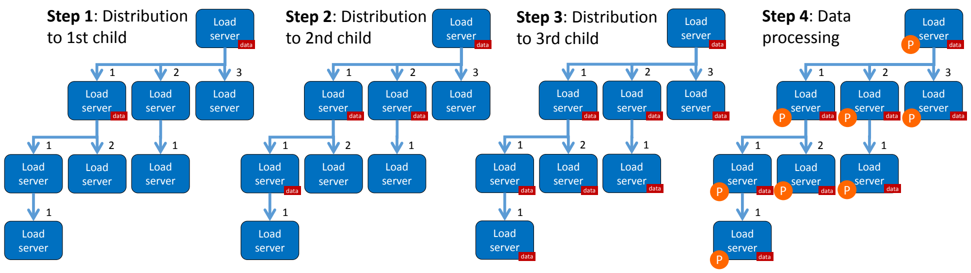

Step 1: Distribution to 1st child
Step 2: Distribution to 2nd child
Step 3: Distribution to 3rd child
Step 4: Data processing
Load server
data
P
1
2
3

Figure 5.7: Spindle network data distribution. Data will be distributed in each node first to the child nodes and then processed on the local node.

5.5 Bootstrapping and Bulk Preloading

During parallel program startup, some actions will be performed on the front-end node. For MPI, for example, the `mpiexec` caller is executed on this node. It parses the call parameters and communicates with run time daemons that are responsible for starting the executable on each assigned compute nodes. The Spindle design takes this activity on the front-end node as part of application startup into account and integrates the node into the overlay network topology by connecting it to the root node of the tree topology. As described in the previous section, LaunchMON and COBO are used to implement the initialization of communication channels, which supports Spindle in implementing two application scenarios: bootstrapping of the executable and bulk preloading. For both, the front-end node has to send local information to the Spindle load servers.

As described in Section 5.2, the executable of the application program will not be loaded by the dynamic linker. Typically, this is done by the startup function of the parallel environment (e.g., `mpiexec`). Spindle can also be used for this purpose. In this case, Spindle will first start a small statically linked executable for each application process (*bootstrapper*), which gets the name and parameter of the real application executable from the front-end node of Spindle. Next, the bootstrapper will send a request to the Spindle load server to move the original executable file to the local disk. Last, the bootstrapper will give the control to the executable with the `exec`-call.

The second application of this feature of Spindle is the bulk preloading. Normally, Spindle loads a library only if at least one of the processes requests it at runtime. A complementary approach is to predict the demand and stage all required libraries in advance. Therefore, Spindle analyzes the executable and extracts its library dependencies. With these, it can initialize the caches of the load servers without waiting for runtime load requests. The subsequent requests for the pre-loaded files are fulfilled immediately. Bulk preloading does not have to cover every possible library. Libraries accessed dynamically, such as those loaded via `dlopen`, can still be loaded through Spindle's push mechanism. In the future, other hints, such as suggestions from the user, can be used to increase the number of preloaded DSOs.

Spindle provides bulk preloading through the front-end client. Since the front-end client is also part of the overlay network, it enables direct communication with the load servers. This allows Spindle to push proactively a collection of libraries to the load servers before the application requests them. On HPC systems where the runtime environment differs between front- and back-end nodes, the paths of preloaded libraries have to be changed to the back-end location. This is supported by the FGFS mechanism (*Fast Global File Status* [2]), a scalable utility to obtain global file properties.

5.6 Python Module Loading

Interpreted languages like Python as a driver of parallel applications do often not only load external binary libraries, they also load (Python) modules, which are represented by small byte-compiled or text-based files required by external modules. These can often be more numerous than shared libraries. In addition to DSOs, Spindle is designed to accelerate the parallel reading of these Python modules, too. When the Python interpreter opens such a file, the client adapter intercepts the attempt using another *rtld-audit* callback function (`*_pltenter`), which is called when an entry in the procedure linkage table (PLT) is resolved. The Spindle client routes these files through the same caching mechanism: they are stored on the local RAM disk and the `open` call is rerouted to their new location. In addition to `open` calls to module files, Python performs also a large number of `stat` calls, which are requesting metadata from the file system. Therefore, the design of Spindle is extended to cache the stat metadata of files, too.

As demonstrated in this section, the *rtld-audit* enables Spindle to intercept various system calls that are issued from the application. This extends Spindle functionality to provide a distributed cache for operations of the dynamic loader to a general cache layer for applications to perform efficient look-up and read operations. Further application scenarios are therefore possible. For example, Spindle can cache common application input files of parallel application.

5.7 Flexible Caching Algorithms

The Spindle design is flexible enough to support a variety of algorithms for request handling, caching, and information forwarding. In its most general form, a client injects search and read requests into the server network. The server forwards the incoming request to the server that is designated as the responsible server for the desired file. Figure 5.8 illustrates the general structure of the Spindle network with five load servers. The load server uses an internal mapping function to find the designated server, which then performs the actual file operation on behalf of the client. The load server should be able to evaluate the mapping function locally without interaction with other load servers. Otherwise, additional traffic will be produced to acquire information from other servers. To fulfill this requirement, the function can, for example, map libraries and directories to load servers by selecting the corresponding server according to the hash value of its file name (or path). Changing the mapping function is one way of configuring Spindle's behavior. The data distribution scheme for results of look-up requests and library data depends on the selected model (pull or push). For the pull model the node that sends a request to another node to obtain file information will also be the destination for the result. Especially the pull model leads to a distributed cache in Spindle, because the data in such a cache structure is distributed among all participating server according to the results of the mapping

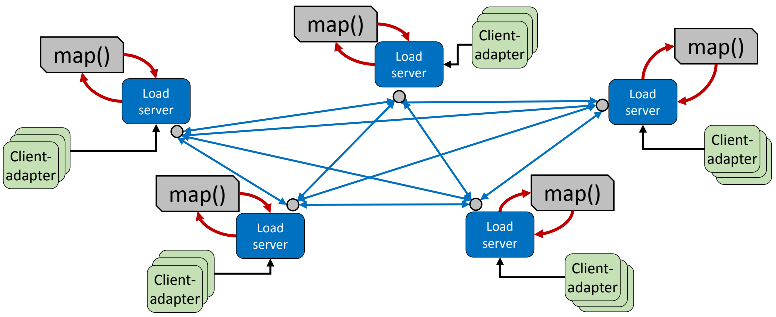


Figure 5.8: Spindle network and mapping-function. In the general architecture of Spindle, each server is able to interact with the file system for a request (not shown). The mapping function `map()` defines the responsible server for file or directory names.

function. The push model requires to broadcast all data directly to all other load servers and implements therefore a replicated cache in Spindle.

In the current implementation, the network topology is a tree. This constrains the communication scheme to a top-down distribution of information from the root node to other nodes (cf. Figure 5.1). In this case, the mapping function is quite simple, because the root server is responsible for all library files. Viewed from the perspective of the file system, the load time of a parallel application is equal to the load time of a single process. The initial implementation assumes that all processes request the same libraries in the same order, which is common for SPMD codes. In this case, a load server does not have to forward an incoming request up the tree. Since the root server will receive all possible requests from its local client, an arbitrary server just has to wait until the root server pushes the desired libraries in its direction. This policy reflects a proactive top-down distribution of cache data, but can cause problems if different processes load different libraries. However, this assumption eased the initial implementation and provided convenient optimizations, but it is not fundamental to the Spindle approach.

For future versions of Spindle it is planned to provide more sophisticated file-distribution schemes. There are at least three use cases that would benefit from more complex distribution schemes. The first one is to use a forest of trees to support efficient access to parallel file systems. Second, a forest of trees can also support the startup phase of MPMD programs and, third, the support for other underlying communication infrastructure like 3d-torus networks using distribution schemes that are adapted to the communication infrastructure.

A communication topology that is built from a forest of trees can take advantages of multiple root nodes that access data from a parallel file system in parallel. Typically, parallel file systems have the capacity to support reading file data from a moderate number of nodes concurrently. Figure 5.9 (right) illustrates how the Spindle network would be configured as a forest of trees. Assuming that the root nodes read their data in parallel, the propagation of data inside individual trees is also accelerated. For example, when replacing a tree containing $n \leq 2^d$ nodes by two sub-trees, the number of nodes in each individual tree will be reduced by a factor of 2 in comparison to a single tree and the maximum distance between the root node and the other nodes will be reduced to $d-1$ hops.

A similar approach can also help for the second use case. As described in Section 1.4.4, applications following the MPMD scheme are running different executables concurrently. This leads to different load sequences of DSOs between these sub-partitions of the application.

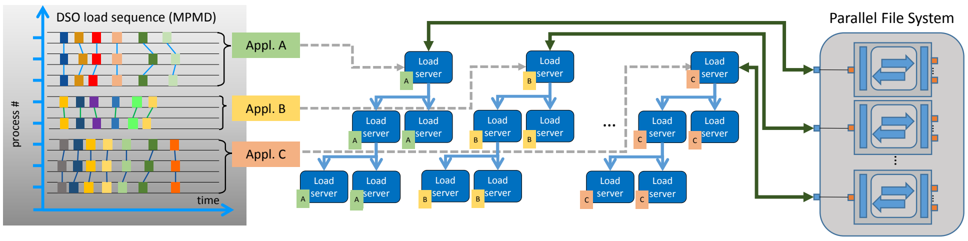


Figure 5.9: Spindle network topology implementing a forest of trees. The different executables of an MPMD application can be mapped on individual trees (center-left). The root nodes of all trees access the parallel file system individually (center-right).

With a forest of trees where each partition maintains its own sub-tree, the load phases of the sub-partitions can be separated. Figure 5.9 (left) demonstrates this optimization for an application that is divided into three partitions running three different executables. A tree root node is then responsible for the requests of only one executable. DSOs and metadata related to the load sequence of this executable are only distributed inside the tree attached to this root node. Compared to a single tree, not only the traffic is separated, also the maximum distance will be reduced by one. This is because a common root node at the top level can be omitted. Tree-to-tree communication is only necessary to a small extend. Therefore, nodes do not need a direct connection to other trees. Instead, only the root nodes of the subtrees will be connected together, for example, with a ring topology.

Third, a flexible network topology for the overlay network gives also advantages on HPC communication infrastructures that uses a 3D- or 6D-torus network. One example is the 3D network of the Cray system, where some of the nodes are selected as I/O routing nodes, which have a direct connection to the file system network. An optimized Spindle network topology would take advantage of this and would select those Spindle load servers as tree root nodes that are on one of the routing nodes or at least close to the routing nodes. Additionally, the nodes could be arranged inside the tree according to number of network hops to the file system. This metric is hardware-related and can be obtained, for example, by the front-end node at application startup. Generally, this approach adapts the overlay network topology of Spindle to the hardware network topology of the HPC system which the application is running on.

5.7.1 Variable network topologies

The bootstrapper COBO implements a single binomial tree topology. The variable network topologies as described in the previous section cannot be implemented by using this software layer. Therefore, a new software layer named *MSocket* was designed as an alternative approach, which allows Spindle to use different network topologies. It implements a fully configurable self-bootstrapping overlay network within the Spindle framework, which allows a free and dynamic definition of the network topology. Similar to COBO, *MSocket* uses internally TCP/IP sockets to implement point-to-point communication channels.

The flexible topology definition of *MSocket* is given with two functions, which have to be specified for each desired topology. The first function will be called at the beginning of the bootstrapping process. It receives a host list of all nodes and computes a connection list, which contains a number of rank pairs, describing each a bi-directional connection in the topology. The rank order given by the host list is obtained on the front-end node or defined by an external layer like LaunchMON. The second function is called during runtime and supports the resolution of routing destinations.

The bootstrapping with *MSocket* is implemented as follows: Each load server is opening a listening port to receive new connection requests. At the beginning of the startup phase, the front-end node connects to the first load server and sends the full contents of the host list to this load server. This server will then compute the connection list using the provided dynamic function. After that, the server will compute for each rank its direct connections to other servers and submit messages containing this information to the other servers. Each load server will establish the connection to the adjacent servers as soon as the content of the message is available. After establishing the connection, the servers are able to route messages according to the results of the provided dynamic routing function. This function has to decide for each message that is not intended for the load server itself to which adjacent server the message has to be forwarded. The function should be able to decide this based on the rank numbers of source node, destination node, and local node.

For example, to implement the binomial tree topology, the connection list can be computed in a similar scheme as the insert operation of priority queues with underlying binomial trees. The insert operation guarantees that keys of the child nodes and its sub tree are larger than the key of the node itself. With this ordering, routing can be decided based on the rank of the child nodes and the destination rank: The message will be forwarded to the child with the largest rank that is smaller than the destination rank.

5.7.2 Optimization for Blue Gene/Q architecture

The network topologies based on COBO and *MSocket* are connecting load servers together. However, the special hardware and system configuration of some large-scale HPC systems do not allow running such a load server on each compute node. One example is the Blue Gene/Q system, where the compute nodes are driven by the reduced Linux kernel and are not equipped with local disk storage. Further, a local RAM disk is also not available on compute nodes and most of the I/O-system calls are forwarded to the I/O nodes. As seen in Section 2.3.3, GPFS supports dynamic loading in a way similar to Spindle: GPFS maintains a big memory cache (page pool), which caches not only files but also metadata information. However, such aggressive caching is not given on BG/Q systems with a Lustre file system. As learned during the tests on Linux clusters with Lustre, metadata requests are forwarded to the Lustre metadata server, which will become a bottleneck at large scale. Therefore, it is expected that dynamic loading on BG/Q with Lustre file systems will have a limited usability. Hence, Spindle is adapted to BG/Q systems as well to support dynamic loading from Lustre file systems.

Due to the limitation on the compute-nodes, Spindle's load server cannot run there. Instead, the load server has to run on the I/O node, which operates a full Linux kernel. However, the number of tasks, which are running on the assigned compute nodes and send requests to the load server on the I/O node, can grow up to 8k processes. The runtime system of Blue Gene/Q addresses this bottleneck by using multiple threads for the I/O daemons on the I/O node to handle the high load of I/O requests. Although this would be possible for Spindle, the load server is not designed as a multi-threaded version of the load server. One reason for this decision is that a multi-threaded daemon would compete with the other daemons on the I/O nodes. Another reason why multi-threading of the daemon is not needed is that Spindle can already take advantage of that typically the processes of SPMP and MPMD programs will start the same executable on one compute node and issue therefore the same sequence of requests for DSOs. So, at least look-up requests can be handled on the compute node locally by exchanging the information directly between the client adapters. Figure 5.10 illustrates how the additional communication layer between the client adapters is implemented. The communication will be established by using a shared-memory segment. The shared segment contains the table for metadata information, which is already requested from the load server. Read and write access to the table is protected by semaphores, in order to serialize the request handling among the

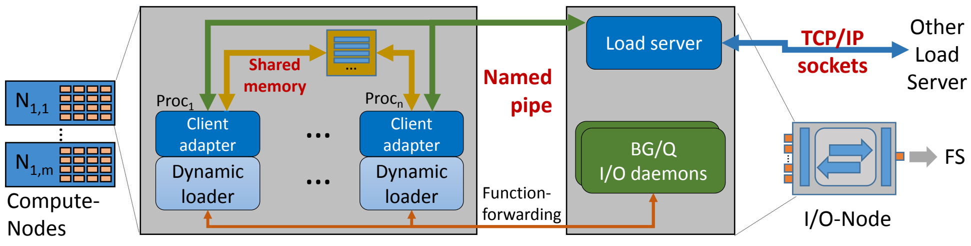


Figure 5.10: Enhanced architecture of Spindle for Blue Gene/Q support with an additional communication layer between the client adapter on the compute nodes.

different client adapters. The first client adapter that does not find a requested metadata entry in the table will send a request for this metadata entry to the load server and will insert the entry in the table as soon it arrives back on the compute node. The connection to the load server is also shared between the client adapters. Therefore, requests are send only once to the load server and the number of connections to the load server is. Compared to a Spindle implementation without local communication, the number of connections is reduced in the case of 8k processes/I/O node by a factor of 64.

Nevertheless, the dynamic loader reads library data furthermore directly from the file system. The I/O calls that are issued by the dynamic loader will be forwarded to the BG/Q I/O daemon on I/O node and executed there. Consequently, library file data of a requested DSO will be transferred multiple times to the compute node. An implementation of a RAM disk on the compute nodes would help only partially, because the runtime system has to be modified to avoid that the I/O calls of the dynamic loader are forwarded to the I/O node.

5.8 Related Work

Spindle is not the first approach that addresses the problem of scalable parallel loading. In this section, we discuss existing solutions to put the work on Spindle into context. The approaches discussed here can be broken down into two categories: those that attempt to improve I/O and storage technologies so that the existing, un-scalable loading algorithm will perform faster, and those that attempt to tackle the root of the problem by modifying the loader to make loading algorithm more scalable.

5.8.1 Parallel file system

A first optimization that can also be done on user level is to stage object code in parallel file systems [45, 95] so that the loader can access it more effectively. This seems like a simple solution to the loading problem. Modern parallel file systems are clusters themselves, which combine multiple disks spread across their nodes into a logical unit. While parallel access to a single disk does not scale well, parallel access to the array of disks scales to the number of nodes in the cluster. Thus, parallel file systems like Lustre [95] significantly improve data-parallel access to large data set files that are striped across the array of disks and where the processes read different parts of the data set. Unfortunately, parallel loading does not exhibit this access pattern.

Parallel loading exhibits no data parallelism, as each process accesses the same, small files. Worse, for library search, parallelism is needed for large numbers of *metadata* operations, and parallel file systems typically have far fewer metadata servers than data servers. As a result, the analysis of dynamic loading on parallel file systems shows that while they offer a performance advantage over NFS when used with a traditional loader, they do not address the fundamental scaling problem in the loader. Coordinating I/O operations in parallel among the loaders themselves can easily outperform NFS or a parallel file system like Lustre.

As demonstrated in Section 2.3.3, GPFS performs besides file data caching also metadata caching in a memory-based page pool. Therefore, less metadata operations like look-up of files in a directory are directly accessing the remote file system server. GPFS implements also replica of metadata blocks, which support parallel access to it. Instead of having one metadata server responsible for all requests, each GPFS client can be the meta-node for a specific file or directory. Therefore, requests are processed from different GPFS clients in parallel. However, file look-ups operations in a directory from a large number of clients can also overload the GPFS client that is responsible for the directory metadata block (cf. Section 1.2.3).

5.8.2 Caching and staging solutions

Many large-scale systems, such as IBM's Blue Gene machines and more recent Cray XT machines, have dedicated I/O nodes that sit closer to the compute nodes than the parallel file system. A common approach for loading on these architectures is staging object code on these I/O nodes and mounting the staging area on the compute nodes. This approach has been used at Argonne National Laboratory (ANL) to accelerate the loading of Python applications on the Intrepid Blue Gene/P machine. It is effective in speeding up the loading process, but it is not transparent to users.

First, it requires application developers to stage their application in a custom I/O node image, which can be tedious when there are large numbers of libraries. Second, users often cannot easily determine which libraries an application needs to load. Most end users of Python, for example, are not familiar with its standard libraries or with those that have DSOs that need to be staged. Third, in some cases the library search path is not known until runtime, so it is impossible to stage *all* DSOs that the application needs.

Cray uses DVS [52, 57], a proprietary I/O forwarding service, to make this process more transparent. DVS forwards file operations via the DVS clients, running on the compute nodes, to the DVS servers that have access to an NFS server using a hierarchy of caches along the I/O forwarding path. However, this approach requires users to stage all application-shared libraries to the dedicated NFS server, which can be tedious — similar to ANL's approach. Further, they do not exploit the full available parallelism of read-only dynamic load routines. Caching directly in the loader is a more direct, coordinated, and scalable approach.

5.8.3 Peer-to-Peer solutions

Dosanjh, et al. propose a peer-to-peer (P2P) architecture for distributing shared libraries across a network [24, 25]. The approach is similar in spirit to Spindle, in that it caches loaded libraries in a RAM disk and aims to satisfy most load requests within the HPC network to reduce file-system load. The approach has the potential for high-bandwidth file distribution, as it is based on the BitTorrent protocol. The authors share the vision of a high-availability parallel loading service; their architecture integrates a loading daemon with the OS. The proposed design requires users to specify all library dependencies in a job description so that they can be seeded to the compute nodes for P2P sharing. This requires, as do the techniques in Sections 5.8.1 and 5.8.2, that users know all library dependencies of their application and specify them in advance. Spindle optimizes the case where dependencies are known in advance, but it is still efficient when dependencies are not known until runtime. In addition, the proposed approach does not address the metadata storm resulting from large-scale application startup. The authors mention that initial seeder processes must handle the first set of requests for libraries as well as a large number of `stat` calls issued by the dynamic loader. However, they do not discuss a coordinated I/O strategy that would allow these seeders to satisfy millions of requests quickly for each job launch. The authors mention distributed hashing techniques, which are promising for scalable P2P loading. These problems are addressed in Spindle with a low-latency tree-based architecture, and by using the *rtld-audit* interface to modify the loader's behavior. The Spindle approach does not require OS daemons and runs entirely in user space.

Solutions that modify the loader

The *collfs* library [14] developed jointly by ANL and the King Abdullah University of Science and Technology (KAUST) provides a scalable dynamic loading service for Python applications. The library allows one process to load libraries, which are broadcast synchronously to the full system. This solution customizes the GNU loader and the Python runtime to use MPI to load libraries. This approach is effective and can drastically speed up many Python programs, but the implementation changes the semantics of loading by requiring that it has to be synchronous. When used this way, some Python programs will require modification so that all loads are performed at the same time, otherwise programs may deadlock.

The Spindle solution handles asynchronous loads of different libraries as well as synchronous loads of the same DSO. The library *collfs* is a good example of how coordinated I/O can speed up loading, but its synchronous semantics and application-specific nature limit its use. It also has technical limitations for system-wide use: it requires a modified version of *glibc*, which makes it very difficult to install. In contrast, Spindle uses the *rtld-audit* interface, which allows Spindle to intercept GNU loader actions without requiring direct modifications to the loader itself. *collfs* also relies on the MPI library, which is not always available at runtime. Spindle works with any programming model for large-scale systems.

5.8.4 Loading as a parallel service

Parallel loading is an example of a case in which a sequential solution applied by each of P processes does not yield good parallel performance. The key observation is that most processes request the *same* objects from the file system, or at least objects that have also been loaded by nearby processes. Rather than accessing the remote file system each time a file is needed, the likelihood that a neighbor has already requested the file should be exploited. Thus, I/O can be coordinated to distribute files much more efficiently and the stress on the limited file system resources can be reduced.

Alternative techniques, like DVS (discussed above), attempt to increase the level of I/O coordination transparently at the file-system level while keeping loader behavior fixed. These techniques have the advantage of maintaining existing abstractions, such as POSIX I/O, which are familiar to users of Unix-like operating systems and which make sense in a sequential environment. However, the strict semantics of such abstractions can limit their scalability in a parallel environment. As an example, POSIX file I/O semantics disallow caching of failed `open` calls, forcing every library search query to go all the way to the file system. Further, traditional file I/O abstractions are oblivious to the *type* of data being transferred, which precludes many parallel optimizations.

In the Spindle approach, the abstraction is raised to the level of the loader, which allows the loader to perform its own coordinated I/O. Thus, it can exploit the knowledge about the files in parallel use. Object code is nearly always read-only and further ample parallelism exists in parallel loading. The performance advantages of exploiting both of these characteristics are too great to be ignored in a parallel environment. For this reason, it is a recommendation that the loader architecture has to be changed for parallel machines. Spindle represents a significant step towards such a truly massively parallel loading service architecture.

As described, Spindle implements a new approach for supporting dynamic loading at large scale, which does not require changes to system software or hardware. Further, executables do not have to be re-compiled or re-linked to be supported by Spindle. The next chapter will show results of measurements with Spindle, which demonstrates its scalability.

6 Evaluation of Parallel Loading with Spindle

As described in the previous chapter, Spindle is designed to reduce the load time of parallel applications that make extensive use of DSOs. To verify this as well as to show the scalability of the described approach, Spindle was tested with two benchmarks, which are running automatically generated codes that load DSOs at program start or during runtime. The first one is *CLoadtest*, a benchmark that generates a pure C-code program and library sources. The second one is the synthetic benchmark named *Pynamic* [61], which is co-designed together with a real application at LLNL to emulate its behavior in a proxy benchmark. The first benchmark program was evaluated on the JSC system JUROPA and was used for the initial performance studies on JUQUEEN; the second one was tested on the Sierra Linux cluster installed at LLNL. The following sections introduce both benchmark programs and present the results running these benchmarks on the two systems JUROPA and Sierra. Finally, the memory footprint of Spindle is discussed.

6.1 Simple Loader Benchmark

Essentially, the simple loader benchmark CLoadtest is a code generator, which produces the benchmark code and the corresponding libraries. The main program and the libraries are generated with pure C-code and have to be compiled before running the benchmark. In this simple benchmark, dynamic loading is only tested for DSOs that are directly linked to the main program. Therefore, only measurements of the application startup phase are possible. Configurations that are more sophisticated like DSO loading at runtime are tested with the Pynamic benchmark, which is described in the next section.

As depicted on the left of Figure 6.1, the code generator can be configured with a number of input parameters: the number of libraries, the code size, the number of functions in a library, and number of C source code files for each library. This set of parameters allows the

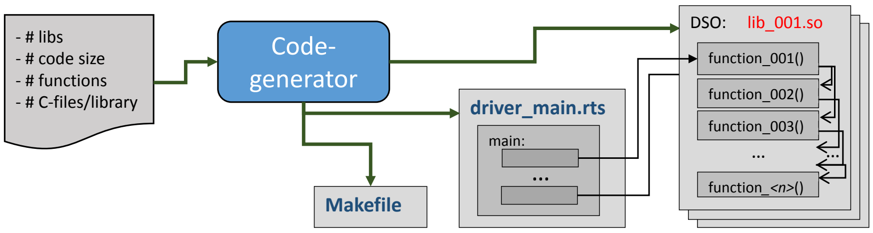


Figure 6.1: The code generator of CLoadtest creates the benchmark program and the DSOs. Libraries and main program are compiled from pure C-code sources.

benchmark to test different aspects of the dynamic loader. For example, on the one hand, large numbers of libraries produce high load on the metadata servers. On the other hand, large code size will generate large library files, which require high bandwidth to the file system (or Spindle caches) during program startup. The third parameter describes the number of functions in each library file. The main program calls only the first function of each library directly. A cascading call tree links together the remaining library functions, so that at the end each function was executed once. The code generator can produce large C files, depending on the selected values for code size and number of functions. To limit the file size and the time to compile a library, the code generator splits the library's source code into multiple files. The number of C files per library is therefore an extra input parameter. In addition to the code files, the generator produces a *Makefile*, which is used to compile all libraries and to generate the dynamically linked program executable. For comparison, the Makefile produces also a statically linked program executable.

The benchmark CLoadtest was used for initial measurements of traditional dynamic loading on JUQUEEN and JUROPA, as described in Section 2.3. In addition, it was also used to evaluate Spindle's performance on the JUROPA system (cf. Section 6.3). Especially, the simple approach of using only libraries with automatically generated C-functions without external dependencies, made it easy to port and run the benchmark on these systems. Furthermore, it has excluded external sources that could interfere with the measurements (e.g., external library files on other file systems or the load of additional files) and therefore, it represents a low-level benchmark for dynamic loading.

6.2 Pynamic

Pynamic is a more advanced benchmark. It supports a configurable emulation of dynamic loading in Python-based applications on massively parallel systems [61]. In this way, Pynamic is a proxy application that closely models the behavior of Python-based multi-physics applications, as they are used on the LLNL systems.

The benchmark uses the scripting language Python together with a parallel MPI environment (*pyMPI* [71]). Similar to the previously described simple benchmark, it uses a code generator to create the Python driver script and DSOs for function libraries and utility libraries. Functions from the first type of libraries are called by methods of the corresponding Python modules. The second type of libraries takes into account that real-world applications often depend on external libraries, which, for example, provide functions for mathematical or physical algorithms. Pynamic has a set of input parameters, which allow users to configure and generate arbitrary numbers of dynamic Python modules and utility libraries of an arbitrary size (cf. Figure 6.2). The number of functions per library is specified as an average value over all libraries and the generator uses randomization to vary this number for individual libraries. Each library has one entry function, which is called by the main driver. Depending on the call depth d, which is an input parameter of Pynamic, the entry function will call every d^{th} function directly; all other functions are called in a cascading chain from there. In addition to this internal dependency, the library function will also call randomly functions from the set of utility libraries.

The main program of the benchmark is a Python script that first imports all library modules and afterwards calls the entry function from each module. The library modules are defined in

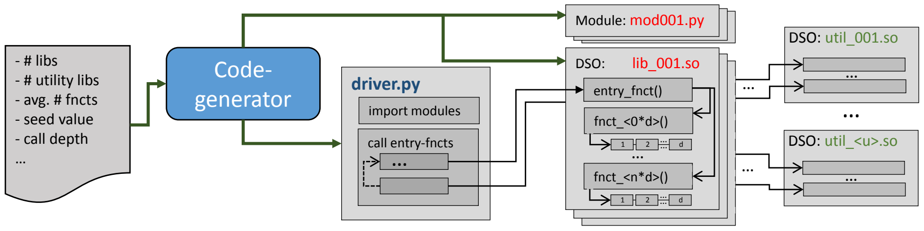


Figure 6.2: The Pynamic code generator creates the Python driver script and the DSOs. In addition, Pynamic generates also a set of utility libraries. Their functions will be called randomly from the functions of the DSOs.

stand-alone Python files, which have the file name suffix *.py*. Internally, Python will look up these files with a similar scheme like libraries, for example, by using a Python search path. After reading the module file, Python will also load the corresponding DSO itself. As an additional action on file system, Python will call the `stat` function on the library file before loading it. As a result, Python generates multiple I/O requests per library, which Spindle has to handled to improve load performance.

The overall measurement results of Pynamic includes three performance metrics, which capture three phases of real world applications using dynamic loading and linking. These are the startup time for the initial executable and library loading, the module-import time for symbol resolution, and the visit time for executing the library functions at runtime. Spindle can reduce the duration of the first two phases in which the program will interact with the file system. However, Spindle cannot change the overhead of process-internal activity like symbol resolution in the third phase. Because Spindle influences the first two Pynamic phases and does not influence the operations in the third phase, the overall runtime of Pynamic was taken as measurements result in the following sections. Pynamic was used for measurements on the LLNL cluster Sierra (cf. Section 6.4).

6.3 Scaling Spindle on JUROPA

JUROPA [55] was until recently the general-purpose Linux cluster at the Jülich Supercomputing Centre. In 2015 the cluster was replaced by the successor system JURECA [54], which is also a general-purpose Linux cluster. Access to the cluster is provided to scientists of Forschungszentrum Jülich, at universities, and at research laboratories in Germany and Europe. It is a resource for small to mid-scale jobs and is in this way complementary to the Blue Gene/Q system as JSC, which is designated to large-scale computing jobs. JUROPA consists of 3288 compute nodes, each equipped with a 2-socket Intel Xeon X5570 CPU (2.93 GHz, Nehalem) and 24 GiB of RAM. The nodes are connected with Infiniband QDR with a non-blocking fat tree topology. On JUROPA, Lustre is used for the scratch and home file systems running together on 14 OSSs within the same interconnect. Therefore, the file systems can be accessed directly over the Infiniband interconnect without using Lustre-specific LNET routers. The scratch file system has 120 Object Storage Targets and a maximum bandwidth of about 50 GiB/s.

All of the following measurements were performed on JUROPA during normal production time. This means that the file system and the interconnect were shared with other jobs during the benchmark runs. Therefore, the measurements were repeated at least three times and the best value was selected. Indeed, with this test scheme, the results showed a variation of load timings, which can be explained with the shared usage of the cluster and the file system. In addition, the JUROPA system does not allow resetting local Lustre memory caches from within a job. Thus, it can be assumed that in the subsequent test runs most of the file data is already available in memory caches. In the first test run it can not be determined how many of the required files are already stored in the memory cache. This is a further reason to select the best of multiple test runs and not the first one. In the next section, we show measurements on the LLNL Sierra cluster, which allow the caches to be cleared at job startup.

The simple loader benchmark was run on JUROPA on up to one third of the system size (1024 nodes) with eight tasks per node in different configurations. Figure 6.3 shows the results for the first test with 710 DSOs and a total library size of 32 MiB. Each library defines three library functions. As already explained in Section 2.3.2, the tests on JUROPA were designed to measure the metadata overhead by using a large number of library files. The first graph (*Dynamic*) shows the initial measurements of dynamic loading without acceleration with Spindle. The time to load the benchmark executable grows linearly with the number of nodes. Therefore, these test runs were stopped at a size of 4096 MPI tasks (512 nodes). As explained in Section 2.3.2, the main reason for this behavior is the serialization of parallel accesses to the Lustre metadata server (MDS). The next curve (*Static*) shows the load time of the statically linked executable. As expected, the load time is very small and increases slightly to 14.3 seconds. Due to the small size of the executable and the simple main program, this curve reflects the system overhead of the initial startup, the MPI initialization, and termination phase. With Spindle support, dynamic loading could be tested on up to 8192 MPI tasks (1024 nodes). The

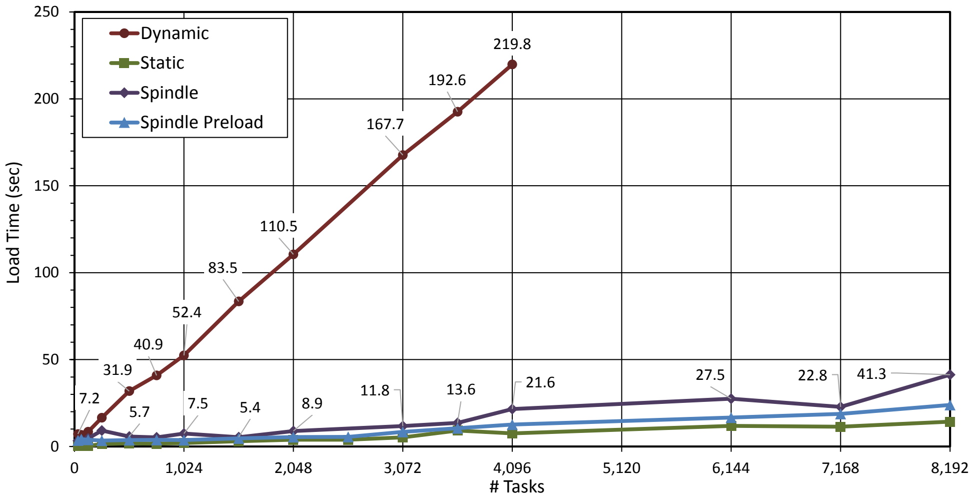


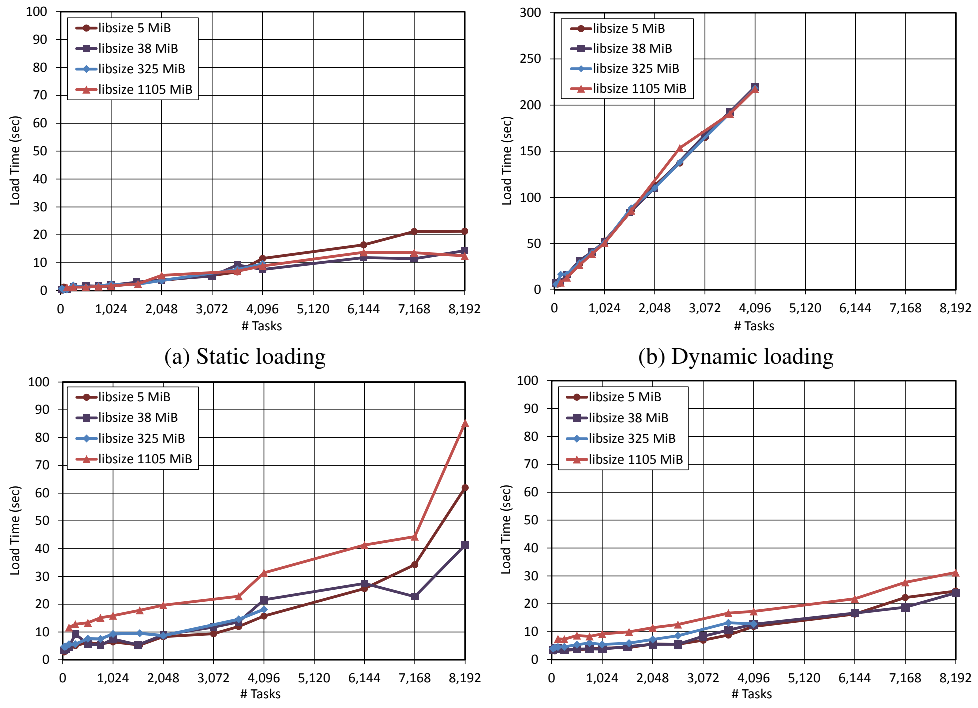
Figure 6.3: Results of CLoadtest benchmark runs on JUROPA loading a statically linked program or a dynamically linked program without and with support of Spindle and preloading (8 tasks per node, 710 libraries, 32 MiB library data).

results show that the load time increases slightly with the number of tasks and remains below 42 sec at the largest scale tested. It is noteworthy that dynamic loading with Spindle support performs at a scale of 4096 MPI tasks (512 nodes) more than ten times faster than traditional dynamic loading.

The last curve (*Spindle Preload*) of Figure 6.3 shows the load time of CLoadtest when using Spindle and *bulk preloading* (cf. Section 5.5). The name and path information of required DSOs was extracted with the Unix-tool `ldd` from the binary file and was passed to Spindle at program start. Spindle then submits the list of libraries to the Spindle servers, which load in this case the libraries to local caches in advance. With this optimization, the load time has been reduced by a factor of 1.7 to a load time of 24 seconds at the largest test scale. For both tests with Spindle, the default *push* model was used. In this case, the top-level Spindle server transfers all library data directly to its children, without waiting on children requests for libraries. Therefore, the data flows down the tree topology directly to the leaf nodes. Without preloading, the top-level server has to wait for a load request for a library from one of the assigned Spindle clients before it can read and transmit the library data. As a result, data transmission is interleaved with the client interaction, which can cause additional waiting time between the transmission steps. In contrast, Spindle with preloading performs all read and transmit operations beforehand in a sequence, which is not interrupted by client interaction. Afterwards, all load requests of Spindle clients can be fulfilled directly by redirecting the client to local copies of the requested libraries. These operations are local to the node and therefore do not lead to additional waiting time.

The remaining extra time of dynamic loading with Spindle and preloading compared to loading a statically linked application is most likely caused by the symbol resolution, which has to be done in the case of dynamic loading during runtime instead of a symbol resolution at link time for static libraries. Another factor is the additional overhead of the so-called `hostbin` startup of Spindle on JUROPA. The `mpiexec` command of the MPI implementation does not support the *MPI_R* interface. Therefore, Spindle cannot use the LaunchMON framework to start the Spindle servers on the application nodes. Instead, Spindle has to start first for each application process a small `hostbin` executable, which then in turn starts the Spindle server and the application process.

In a second test, the benchmark program was run with a set of libraries with different sizes to determine how the amount of library data affects Spindle. The previously described issue of Lustre caching on local nodes can be ignored in this case: Spindle reads library data only on one node from the Lustre file system, whereas all other nodes get their data via interconnect from this node without file-system interaction. Figure 6.4 shows results of the measurement for overall library sizes from 5 MiB to 1.1 GiB using the four different load schemes. The number of libraries and the number of library functions remain unchanged compared to the previous test. As expected, the size of the executable does not influence the time for loading the statically linked executable and for traditional dynamic loading. The observed variation of load timings is most likely due to the influence of activities of other jobs on the system. Even with the local caching of library data, the load time for the traditional dynamic loading increases significantly with the number of tasks, which is independent of the library data size. This indicates again that look-up operations of the Linux loader are the main bottleneck on the metadata server. Both measurements with Spindle support indicate that the overall library



(a) Static loading

(b) Dynamic loading

(c) Dynamic loading, Spindle

(d) Dynamic loading, Spindle/preloading

Figure 6.4: CLoadtest benchmark on JUROPA with varying library sizes and different load schemes (8 tasks per node).

size does not directly influence the load time. Similar to the previous test with small library sizes, the variations of the load timing are much larger without preloading than the timings with bulk preloading. One exception is the measurement with a total library size of 1.1 GiB. Possible reasons are limitations of the capacity of memory cache and the saturation of network traffic on individual links of the interconnect. In the latter case, a mapping of the Spindle tree topology of the overlay network onto the interconnect topology (fat tree) would help, which will be a topic of future investigation.

The results of both tests on JUROPA demonstrate that Spindle enables dynamically linked applications to run at large scale. Especially, Spindle solves the metadata bottleneck of the parallel file system Lustre when performing large number of file tests in parallel.

6.4 Scaling Spindle on Sierra

The Sierra Linux cluster is utilized at Lawrence Livermore National Laboratory as a workhorse for solving computationally intensive problems. It is equipped with 1,856 compute nodes. The nodes have a 2-socket Intel Xeon EP X5660 CPU (2.8 GHz, Westmere) with 12 cores and 24 GiB of RAM. Nodes are connected by a Qlogic Infiniband QDR interconnect. Two file systems are available on Sierra: NFS and Lustre. NFS is used for the home file system and Lustre is used for scratch data space. Similar to JUROPA, all tests ran within the normal production environment, the system was shared with other concurrently running jobs. Therefore, measurements were repeated multiple times, and results were taken from the best run. In all of these tests, Pynamic was configured to load 495 shared objects, a total of 1.1 GB of library files.

In the first test, the Pynamic benchmark was run without Spindle and with loading libraries from NFS. As shown in Figure 6.5, the overall runtime increased rapidly. The measurement was stopped at a small scale (< 1200 processes) to prevent an I/O storm, which would have affected other users on the system. The runtime increased more than linearly. The poor parallel support of NFS can explain this.

As described in Section 5.8.1, a parallel file system is better suited for the types of file operations used in Pynamic. Therefore, the same test was run on Sierra with loading library files from the Lustre file system. Figure 6.5 shows a nearly linear growth in the runtime on Lustre, which allowed to run the tests on up to 6,144 processes (512 nodes). The linear scaling of these runs shows significant improvement compared to prior experiences with Lustre on Sierra. Presumably, the relatively recent enabling of read caches on the Lustre servers is the reason for this behavior. Further improvements could be expected if Lustre was better configured for the type of I/O access pattern associated with Pynamic (small files and high metadata rate), although this is not the typical file access pattern for HPC file systems (large files and data parallelism). The measurements show that Lustre could be a partial solution to the loading problem, but only up to a moderate number of processes. The Lustre performance was already starting to degrade at these scales, and would likely have suffered more at larger scales.

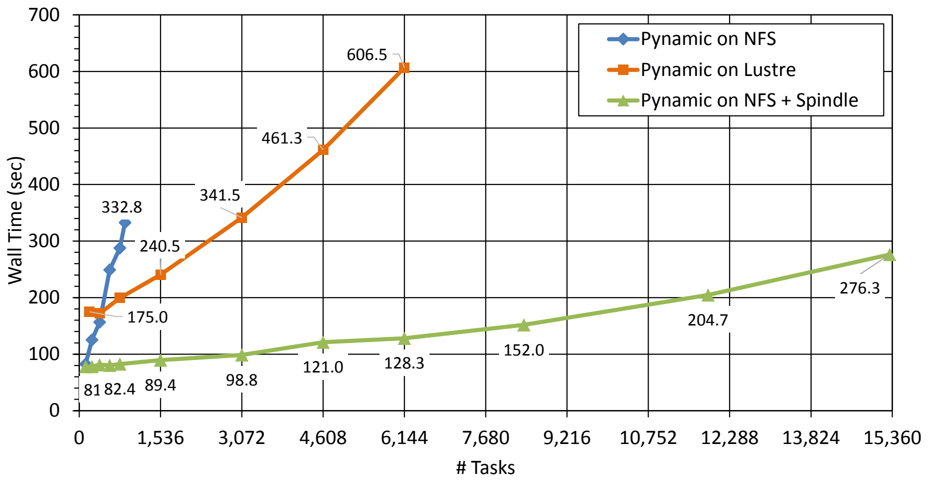


Figure 6.5: Pynamic benchmark run on Sierra with NFS and Lustre compared to Pynamic with NFS and using Spindle (12 tasks per node, 495 shared objects, 215 utility libraries, 1.1 GiB library data).

In the third test case, Pynamic was run with Spindle, with libraries hosted on the NFS file system. As shown in Figure 6.5, Spindle allows Pynamic to be run at the largest allocation size on Sierra, with 15,312 processes distributed over 1,276 compute nodes. Spindle reduced the overall Pynamic runtime to 276 seconds. This is faster than the NFS test with 64 nodes and the Lustre test with 256 nodes.

Spindle allowed to run Pynamic at a significant larger scale. However, there is an unexpected growth in the total runtime, from 81 seconds at a small scale to 276 seconds at higher scales. Pynamic's built-in timings did not help to distinguish whether this increase in time was due to poor scaling in Spindle or whether it was part of Pynamic's internal activity. To get a better understanding of this effect, tests were repeated using the bulk-preloading feature of Spindle, which separates the library load process from the benchmark execution (cf. Section 5.5). In this mode, the front-end process reads the list of prerequisite libraries and instructs the load server to load them into their caches before Pynamic starts. This allows the time for Spindle data distribution to be measured separately and not as part of the Pynamic overhead. Figure 6.6 shows the results of this measurement. The time for bulk preloading seems constant, although in theory it should show logarithmic growth. The maximum number of message transfer hops from the root node to the leaf nodes in the tree topology of Spindle overlay network grows from eight hops at 128 nodes to twelve hops at 1,276 nodes (cf. Section 5.4). Furthermore, the Spindle tests at larger scale were performed with multiple hand-coded trees, which limited the depth of each individual tree and the time to broadcast data (cf. Section 5.7). Given that the total Pynamic runtime continued to grow, even with all files pre-staged to the RAM disk, it can be concluded that this growth is caused by other internal activities of Pynamic. Besides the communication as one of the possible reasons, a `stat` system call is issued in the Python module loading process for each library, which accesses, similar to the `open`-calls,

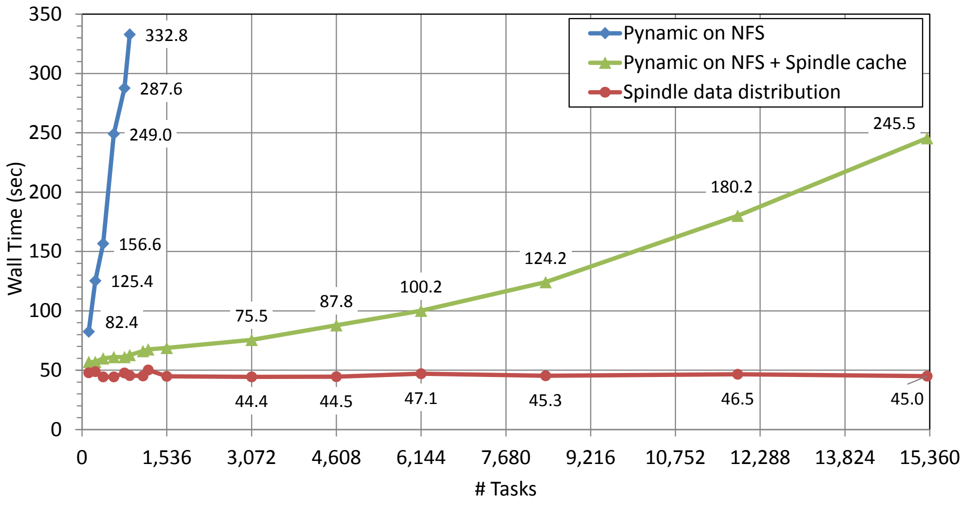


Figure 6.6: Pynamic benchmark run on Sierra with NFS and Spindle bulk preloading showing the time data distribution and startup from a pre-loaded Spindle cache separately. As comparison is shown the time for Pynamic with NFS only (12 tasks per node, 495 shared objects, 215 utility libraries, 1.1 GiB library data).

the file system too. In the meantime, Spindle has been developed further and in the current version the Spindle clients handle also these calls. Unfortunately, the measurements, which were performed in the first development phase, could not be repeated with the new version of Spindle due to operational access restriction on Sierra.

6.5 Memory overhead of Spindle

As discussed in Section 5.3.2, the Spindle memory overhead depends on the portion of code pages that an application normally maps into memory. Spindle causes the application to use memory as if 100% of its code pages were always resident. This overhead was measured on a single node of the LLNL cluster Sierra using Pynamic, configured to load 495 libraries, which produced an application with 488 MiB of loadable code. To do this, the version of Pynamic used was modified to touch a certain part of its memory pages artificially, loading them into its working set. This allowed to compare Spindle memory overheads for different working set sizes. The amount of memory used was measured by running Pynamic, writing data to each page and then checking how much memory could be allocated before the system began paging out in order to make room for the new allocations.

Figure 6.7 illustrates two aspects of the memory overhead of Spindle. First, it shows that the memory overhead is predominately a factor of the application's working set size. In the cases, where the application has only a small working set, Spindle loads many unneeded pages into memory via the RAM disk and produces an overhead in the order of the application's size. In the cases, where the application actually uses most of its code, the RAM disk shares most of its physical memory with the process and the amount of memory available does not change significantly. Second, Figure 6.7 shows that the memory overhead of the client and server is relatively small. The data point at 100% represents the case where the RAM disk memory is completely shared with the virtual memory of the process. The remaining overhead of 15.2 MiB is approximately equal to the memory used by the client and the server.

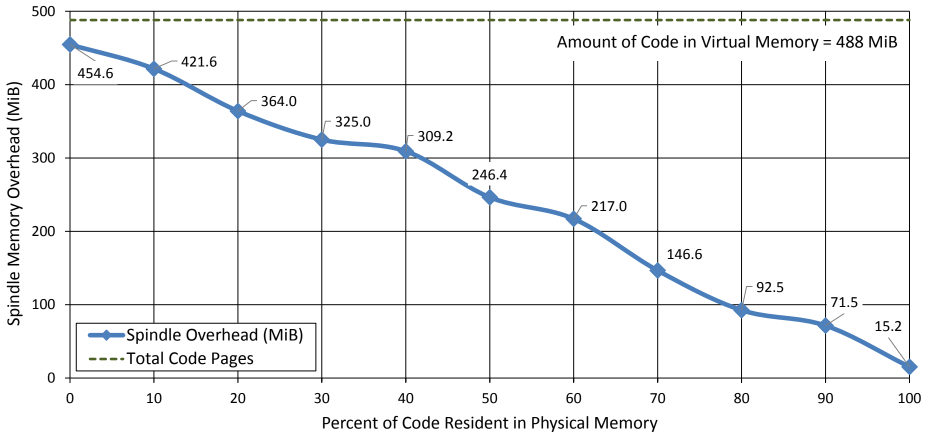


Figure 6.7: Memory overhead of Spindle when running Pynamic on a node of Sierra (495 shared objects, 488 MiB library data).

7 Conclusion & Outlook

This chapter summarizes the concepts and techniques for parallel task-local I/O and the loading of dynamically linked applications developed in this dissertation. Furthermore, the current states of the two tools SIONlib and Spindle, which implement these techniques, are discussed. Finally, the chapter provides an outlook for each approach and speculates about further development based on the results of this work.

Task-Local I/O with SIONlib

The first part of this dissertation addresses a common scalability problem of traditional parallel task-local I/O on large-scale systems: the very high overhead of file creation and the difficulty of managing excessive numbers of files. These limitations have motivated the development of the I/O library SIONlib. It solves the two problems by transparently mapping a large number of task-local files onto a very small number of physical files and by implementing a method to handle internal metadata. In this way, the time needed for the parallel creation of hundreds of thousands of task-local files can be reduced from several minutes to just a few seconds. With the introduction of shared file I/O, new challenges appeared. For example, to avoid concurrent access to file-system blocks, which results in file-locking overhead, SIONlib aligns data chunks to the boundaries of these file-system blocks in its internal data format. Furthermore, the parallel access to a shared file creates another metadata bottleneck at larger scale. To address this, SIONlib implements a multi-file approach, which separates tasks of parallel applications into smaller sub-groups; each creating its own shared file to distribute metadata management across multiple I/O components. This approach could serve as one of the key solutions to the implementation of high scalable I/O on exascale systems.

As demonstrated with benchmarks and application scenarios, the approaches described in this dissertation enable applications, with the support of SIONlib, to perform task-local I/O up to a scale of 1.8 million tasks. This is achieved while maintaining a high bandwidth of 60-80 % of file-system peak bandwidth without paying the penalty of long file creation times. A final key advantage of SIONlib is that it operates fully in user space and requires very few source code changes in applications to make use of the library. 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 [85].

Additional features of SIONlib address special use cases of parallel task-local I/O. For example, the coalescing I/O feature enables those applications that write and read small data blocks from a large number of tasks to benefit from SIONlib. As demonstrated with the evaluation of parallel task-local I/O in the application MP2C and also recently with the KKRnano code, applications with such small I/O requirements per tasks benefit considerably from this approach.

Further new features of SIONlib, such as the support for key-value containers and the processing of SIONlib files from parallel programs with a different number of tasks (mapped open), enable parallel tools like Scalasca, Score-P, and Vampir, which perform internally task-local I/O, to benefit from the approaches of SIONlib.

Shared I/O to a file container as implemented by SIONlib imposes some restrictions on the application. For example, the maximum amount of data written or read in one piece has to be known in advance. This restriction can be lifted with the planned changes to SIONlib API in the next major release. The use of the write and read wrapper functions will be mandatory. This will improve the management of data blocks within the file container considerably. Another limitation of the SIONlib file layout is that the maximum number of tasks must always be known in advance. This is especially challenging for dynamic process management. SIONlib addresses this restriction by using key-value containers, which allow the library to store different data segments in a file container without any limitation in number. Since currently only parallel tools require this feature, it has to be analyzed whether parallel applications can also benefit from key-value containers. If so, additional optimization and adaptation of this feature for use in parallel applications may be needed. The technique of using multiple physical files for a file container also enables the use of node-local or partition-local storage for managing these files, namely because SIONlib does not require a global name space. For example, this allows users on the K computer to exploit local storage in its file system. More evaluation of SIONlib on such I/O architectures will be undertaken in the near future.

Further work on SIONlib within the EU project DEEP-ER [88] will add functionality to exploit local storage for efficient checkpointing. This allows, in combination with the multi-level checkpoint library SCR [73], to establish higher levels of resiliency. SIONlib is already integrated into a number of applications, including the mesoscopic hydrodynamics code MP2C [30, 89], the parallel coulomb solver PEPC [39], the highly scalable program muphi [92] for simulation of flow and transport in porous media, and the simulator for spiking neural network models NEST [81]. The integration of SIONlib into further applications will continue within the EU project EoCoE (Energy-oriented Centre of Excellence), which started in October 2015. As the number of large-scale applications that use SIONlib for parallel task-local I/O grows, SIONlib is gaining further attention. One recent example is the report "Storage Systems and Input/Output to Support Extreme Scale Science" sponsored by the US Department of Energy (DOE), which already cites SIONlib as a state-of-the-art I/O library [77].

Putting this work in a broader perspective, we believe that the observations regarding the scalability of traditional parallel task-local I/O will raise the awareness of this problem in the wider HPC community. Furthermore, we hope that the general ideas developed in this work to implement a scalable solution for parallel task-local I/O will ultimately be adopted by designers of standard file systems, I/O infrastructures, and high-level I/O libraries. SIONlib represents in this way a reference implementation for efficient parallel task-local I/O.

Dynamic loading with Spindle

The second part of this dissertation addresses the scalability of loading dynamically linked applications. Dynamic linking and loading have gained recognition in HPC due to their advantages for managing growing application complexity. However, the mechanisms used to load shared objects today do not scale to the level required by larger supercomputers. The tasks of locating and then loading a dynamic shared object involve many file-system operations. When a large number of processes load many DSOs simultaneously, the resulting I/O storm can disrupt even the largest parallel file system. As shown by experience, loading of such applications does not scale further than 512 compute nodes on typical Linux clusters. The main reason for this limitation is the serial design of the dynamic loader, a component that searches and loads the dynamic libraries at program startup. These search and load operations are done independently by each process, although in parallel applications all processes load a similar set of DSOs and the look-up and search operations occur during a short time interval at program startup.

Spindle addresses these critical challenges by extending the dynamic loader to exploit the similarity of load operations of parallel applications for the coordination of the file-system operations. Therefore, three main techniques are applied. First, the look-up and load operations of the GNU dynamic loader are transparently intercepted via the audit interface in user space. Second, instead of operating on the file system, they interact now with the Spindle server, which is running side-by-side with the application. Finally, an overlay network connects the servers and in this way allows the implementation of a distributed cache for library files and their metadata. Spindle is working transparently with applications and all components are running in user space. In addition, modifications to the runtime system are not necessary and users are not required to make changes to their applications or work flows. With the presented techniques, Spindle reduces the number of file-system operations for a parallel application to the number of operations for a single program. The experiments with two benchmarks on the Linux clusters at LLNL and JSC, both equipped with a Lustre file system, show that Spindle is highly scalable with limited memory and performance overhead. Although the implementation of these techniques is very new and some engineering efforts to port, optimize, and customize Spindle are expected, it is imperative to provide Spindle to a broader range of platforms and applications. Thus, Spindle is available to the community for download under an open-source license [87].

Spindle is already capable of addressing scalability challenges in current production environments like the Linux clusters at LLNL and JSC and has no apparent scalability limit in sight. The scalability of Spindle could be demonstrated by running the benchmark Pynamic, a proxy benchmark for a real application on the Sierra cluster at LLNL at large scale. The results show that the startup of dynamically linked applications is now feasible on more than 15,000 tasks, whereas the overhead of Spindle is nearly constantly low. Similar results could be achieved with another benchmark on the JUROPA at JSC. First experiments on the Jülich Blue Gene/Q system JUQUEEN show that the GPFS file system together with the hierarchical infrastructure of Blue Gene/Q provides some kind of implicit caching of file metadata. However, as indicated by measurements on JUQUEEN on a NFS based file system, this caching is not provided on non-GPFS systems. Therefore, the port and optimization of Spindle to the Blue Gene/Q system with its hierarchical I/O infrastructure is the next step in the road map. Further optimizations of Spindle will include the automatic reduction of data exchanged among load servers. For example, Spindle could check whether a requested library is a system library. System libraries may reside in a node-local file system, where caching is not needed. Finally, it is planned to improve server efficiency through multithreading to allow local and network operations to be performed concurrently.

Spindle implements a transparent solution for the loading of dynamically linked applications at scale. Furthermore, Spindle represents a transient solution as it mitigates the limitations of current system components, which were initially developed for a serial environment. Looking at the complexity of future systems, the basic ideas and the architecture of Spindle will help to pave the way for a massively parallel OS/runtime loading service for future exascale machines.

Contribution to Publications

The work on the approach to improve parallel task-local I/O at large scale was begun in 2008. The first version of SIONlib, which implements this approach, has been integrated into Scalasca v1.0 [35, 93, 94]. The principles, the implementation, and first results were presented in 2009:

> Wolfgang Frings, Felix Wolf, and Ventsislav Petkov. Scalable Massively Parallel I/O to Task-Local Files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC'09*, pages 17:1–17:11, New York, NY, USA, 2009. ACM. [33]

In 2011, further optimization and measurement results on the JUGENE system were reported in the following paper:

> Wolfgang Frings and Michael Hennecke. A system level view of Petascale I/O on IBM Blue Gene/P. *Computer Science - Research and Development*, 26:275–283, 2011. 10.1007/s00450-011-0154-4. [32]

SIONlib is under ongoing development. The recently developed methods for *mapped open*, storage of *key-value pairs* and the design of the *generic interface* are work within a BMBF funded project on Score-P [59]. SIONlib is integrated into Score-P for applications running with different programming models and included in the latest available release of Score-P (2015). Further development of SIONlib in the direction of resiliency will be done within the EU project DEEP-ER [88]. SIONlib is already integrated into a number of applications, for example, MP2C [89, 30], PEPC [39], muphi [92] and NEST [81]. The integration of SIONlib into further applications will also be continued within the EU project EoCoE (Energy-oriented Centre of Excellence, started Oct. 2015).

The research on the second approach to improve dynamic loading at large scale was begun in 2011 and was performed by the author during a research visit to Lawrence Livermore National Laboratory (LLNL) in 2012. The developed methods were combined to create the tool Spindle and benchmarked at this time. The results were published in 2013 and received the **best paper award** of the ICS'13 conference:

> Wolfgang Frings, Dong H. Ahn, Matthew LeGendre, Todd Gamblin, Bronis R. de Supinski, and Felix Wolf. Massively Parallel Loading. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 389–398, New York, NY, USA, 2013. ACM, Best Paper Award. [31]

Since the research visit in 2012, the author has been working together with LLNL on Spindle. One of the LLNL goals is to make Spindle to an easy-to-install open source project by adding an automatic configuration and build system. In addition, LLNL added code to secure the internal communication. One of the future goals of the author will be to focus on communication strategies for Blue Gene/Q.

Glossary

ccNUMA cache-coherent Non-Uniform Memory Architectures.

CN compute node.

DSO Dynamic Shared Object.

ELF Executable and Linkable Format, used on most Unix-like systems to store program executables.

ethernet channel or *EtherChannel* is primarily deployed on Cisco-switches. It aggregates the available bandwidth of multiple physical ethernet links by grouping these into one logical ethernet link.

false sharing describes a memory access pattern of multi-threaded applications on systems with coherent cashes. It degrades the performance of such applications [10].

FEFS Fujitsu Exabyte File System, is scalable parallel file system based on Lustre and is used for the K computer at Riken.

GiB gibibyte, $2^{30} = 1024^3$ bytes (cf. IEC 80000-13 [29]).

GNR GPFS native RAID feature.

GOT Global Offset Table.

GPFS General Parallel File System, is a commercial parallel file system from IBM.

GSS IBM System x GPFS Storage Server, which implements RAID functionality on software level in GPFS.

HDF Hierarchical Data Format.

HDF5 Hierarchical Data Format (version 5).

ION I/O node, e.g. in Blue Gene/Q system used to forward I/O requests to the file system.

JSC Jülich Supercomputing Centre.

KiB kibibyte, $2^{10} = 1024^1$ bytes (cf. IEC 80000-13 [29]).

lazy binding describes the deferred symbol binding during runtime.

LLNL Lawrence Livermore National Laboratory.

Lustre is a open-source parallel file system for large-scale system, which implements a distributed object based storage.

MDS Meta Data Server, component of the Lustre parallel file system.

MDT Meta Data Target, stores metadata information about files and directories of the Lustre parallel file system.

MiB mebibyte, $2^{20} = 1024^2$ bytes (cf. IEC 80000-13 [29]).

MIMD *multiple instruction multiple data*, execution model following Flynn's classification of parallel architectures [27].

MIMD multiple instruction multiple data.

MPI *Message Passing Interface*, a parallel programming paradigm, designed for distributed-memory systems.

MPMD multiple program multiple data.

MPP parallel multiprocessor.

NFS Network File System.

NUMA Non-Uniform Memory Architectures.

obfds object-based disk file system, predecessor of the Lustre file system.

OpenMP *Open Multi-Processing*, a parallel programming paradigm, designed for shared-memory systems.

OSS Object Storage Server, component of the Lustre parallel file system.

OST Object Storage Target, maintain a local files system to store data of the Lustre parallel file system.

OTF2 Open Trace Format 2.

PHDF5 Parallel HDF5 library.

PiB pebibyte, $2^{50} = 1024^5$ bytes (cf. IEC 80000-13 [29]).

PIC Position-Independent Code.

PIOFS Parallel I/O File System, was designed for the parallel computer IBM SP2 and is the predecessor of GPFS, a commercial parallel file system from IBM.

PLT Procedure Linkage Table.

RAID Redundant Array of Independent Disks.

RAID-6 Level 6 of RAID using block-level striping with double distributed parity.

RDMA Remote Direct Memory Access is often used in HPC network design to optimized the communication between components of a HPC system. RDMA implements a one-sided-communication and supports zero-copy.

SMP Symmetric Multi-Processor.

SPMD single program multiple data.

TiB tebibyte, $2^{40} = 1024^4$ bytes (cf. IEC 80000-13 [29]).

UMA Uniform Memory Architectures.

working set describes the pages of an application process, which are used during run-time and are therefore stored in memory.

Bibliography

[1] Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, Adam Moody, and Martin Schulz. Overcoming scalability challenges for tool daemon launching. In *The International Parallel and Distributed Processing Symposium*, Boston, MA, 2013.

[2] Dong H. Ahn, Michael J. Brim, Bronis R. de Supinski, Todd Gamblin, Gregory L. Lee, Matthew P. LeGendre, Barton P. Miller, Adam Moody, and Martin Schulz. Efficient and scalable retrieval techniques for global file properties. In *The International Parallel and Distributed Processing Symposium*, Boston, MA, 2013.

[3] Nawab Ali, Philip Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert Ross, Lee Ward, and P. Sadayappan. Scalable I/O forwarding framework for high-performance computing systems. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[4] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. FTI: high performance Fault Tolerance Interface for hybrid systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 32. ACM, 2011.

[5] Eli Bendersky. Articles about "Linkers and loaders", 2010-2013. `http://eli.thegreenplace.net/tag/linkers-and-loaders` (visited Jan 2015).

[6] John Bent. *High Performance Parallel I/O*, chapter PLFS: Software-Defined Storage for HPC, pages 169–176. Computational Science Series. Chapman & Hall, CRC Press, 2014.

[7] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a Checkpoint Filesystem for Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 21. ACM, 2009.

[8] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 4.0*, July 2013.

[9] David Böhme, Felix Wolf, Bronis R. de Supinski, Martin Schulz, and Markus Geimer. Scalable Critical-Path Based Performance Analysis. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1330–1340. IEEE, 2012.

[10] William J Bolosky and Michael L Scott. False sharing and its effect on shared memory performance. In *Proceedings of the Fourth symposium on Experiences with distributed and multiprocessor systems*, 1993.

[11] Peter J. Braam et al. The Lustre Storage Architecture. Technical Report, Cluster File Systems, Inc. 2003.

[12] Michael J. Brim and Barton. P. Miller. Group File Operations for Scalable Tools and Middleware. Technical Report No. TR1638, Computer Sciences Department, University of Wisconsin, 2008.

[13] Dirk Brömmel, Ulrich Detert, Stephan Graf, Thomas Lippert, Boris Orth, Dirk Pleiter, Michael Stephan, and Estela Suarez. Paving the Road towards Pre-Exascale Supercomputing. In *Proceedings of NIC Symposium 2014, Jülich, Germany*. Forschungszentrum Jülich, 2014.

[14] Jed Brown, William Scullin, and Aron Ahmadia. Solving the import problem: Scalable Dynamic Loading Network File Systems. Talk at SciPy conference, Austin, Texas, July 2012. `http://pyvideo.org/video/1201` (visited Jan 2015).

[15] Huy Bui, Hal Finkel, Venkatram Vishwanath, Salma Habib, Katrin Heitmann, Jason Leigh, Michael Papka, and Kevin Harms. Scalable parallel I/O on a Blue Gene/Q supercomputer using compression, topology-aware data aggregation, and subfiling. In *22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 107–111. IEEE, 2014.

[16] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, chapter 2.2. Wiley, Chichester, UK, 1996.

[17] Franck Cappello, Al Geist, William Gropp, Sanjay Kaleand Bill Kramer, and Marc Snir. Toward Exascale Resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014. Open Access, `http://superfri.org/superfri/article/view/14` (visited Jan 2015).

[18] Juelich Supercomputing Centre. JSC-Workshop "Blue Gene Active Storage", 12 2013. `http://www.fz-juelich.de/ias/jsc/EN/Expertise/Services/Documentation/presentations/presentation-bgas_table.html`, (visited Dec 2014).

[19] PMGR Collective Startup. `http://sourceforge.net/projects/pmgrcollective/` (visited Jan 2015).

[20] T. I. S. Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, May 1995.

[21] IBM Corporation. Jülich Supercomputing Center tackles the grand challenges of research. `http://public.dhe.ibm.com/common/ssi/ecm/xs/en/xsc03152usen/XSC03152USEN.PDF` (visited Jan 2015).

[22] Robert C. Daley and Jack B. Dennis. Virtual memory, processes, and sharing in MULTICS. *Commun. ACM*, 11(5):306–312, May 1968.

[23] Juan Miguel Del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved Parallel I/O via a Two-phase Run-time Access Strategy. *ACM SIGARCH Computer Architecture News*, 21(5):31–38, 1993.

[24] Matthew G. F. Dosanjh, Patrick G. Bridges, Suzanne M. Kelly, and James H. Laros III. A Peer-to-Peer Architecture for Supporting Dynamic Shared Libraries in Large-Scale Systems. In *Fifth International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, 2012.

[25] Matthew G. F. Dosanjh, Patrick G. Bridges, Suzanne M Kelly, James H. Laros III, and Courtenay T. Vaughan. An evaluation of BitTorrent's performance in HPC environments.

In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, page 8. ACM, 2014.

[26] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible Hashing – A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems (TODS)*, 4(3):315 – 344, 1979.

[27] Michael Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[28] Fraunhofer ITWM Competence Center for High Performance Computing. The high-performance parallel file system BeeGFS. `http://www.fhgfs.com/cms/` (visited Dec 2014).

[29] International Organization for Standardization. ”IEC 80000-13:2008”, ”Quantities and units”, Part 13 ”Information science and technology” , 2008. `http://www.iso.org/`.

[30] Jens Freche, Wolfgang Frings, and Godehard Sutmann. High-Throughput Parallel-I/O using SIONlib for Mesoscopic Particle Dynamics Simulations on Massively Parallel Computers. In *Advances in Parallel Computing*, volume 19, 2010, pages 371 – 378, 2010.

[31] Wolfgang Frings, Dong H. Ahn, Matthew LeGendre, Todd Gamblin, Bronis R. de Supinski, and Felix Wolf. Massively Parallel Loading. In *Proceedings of the 27th ACM International Conference on Supercomputing*, ICS '13, pages 389–398, New York, NY, USA, 2013. ACM. (ICS 2013 Best Paper).

[32] Wolfgang Frings and Michael Hennecke. A system level view of Petascale I/O on IBM Blue Gene/P. *Computer Science - Research and Development*, 26:275–283, 2011. 10.1007/s00450-011-0154-4.

[33] Wolfgang Frings, Felix Wolf, and Ventsislav Petkov. Scalable Massively Parallel I/O to Task-Local Files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 17:1–17:11, New York, NY, USA, 2009. ACM.

[34] Kui Gao, Wei-keng Liao, Arifa Nisar, Alok Choudhary, Robert Ross, and Robert Latham. Using Subfiling to Improve Programming Flexibility and Performance of Parallel Shared-file I/O. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 470–477. IEEE, 2009.

[35] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Daniel Becker, David Böhme, Wolfgang Frings, Marc-André Hermanns, Bernd Mohr, and Zoltán Szebenyi. Recent Developments in the Scalasca Toolset. In *Tools for High Performance Computing 2009*, pages 39–51. Springer Berlin Heidelberg, 2010.

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

[37] Michael Gerndt, Karl Fürlinger, and Edmond Kereku. Periscope: Advanced Techniques for Performance Analysis. In *Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, pages 15–26. John von Neumann Institute for Computing, Jülich, NIC Series, Vol. 33, ISBN 3-00-017352-8, 2005.

[38] Mark Giampapa, Tom Gooding, Todd Inglett, and Robert W. Wisniewski. Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1 –10, Nov. 2010.

[39] Paul Gibbon, Robert Speck, Anupam Karmakar, Lukas Arnold, Wolfgang Frings, Benjamin Berberich, Detlef Reiter, and Martin Mašek. Progress in mesh-free plasma simulation with parallel tree codes. *IEEE Transactions on Plasma Science*, 38(9):2367–2376, 2010.

[40] Megan Gilge et al. *IBM System Blue Gene Solution Blue Gene/Q Application Development*. IBM Redbooks, 2014. `http://www.redbooks.ibm.com/abstracts/sg247948.html` (visited Jan 2015).

[41] Gary Grider, Lee Ward, Robert Ross, and Garth Gibson. A Business Case for Extensions to the POSIX I/O API for High End, Clustered, and Highly Concurrent Computing, 2006.

[42] The HDF Group. Hdf5. `http://www.hdfgroup.org/HDF5/` (visited Jan 2015).

[43] Dean Hildebrand, Arifa Nisar, and Roger Haskin. pNFS, POSIX, and MPI-IO: a tale of three semantics. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, pages 32–36. ACM, 2009.

[44] Dean Hildebrand and Frank Schmuck. *High Performance Parallel I/O*, chapter GPFS, pages 107–117. Computational Science Series. Chapman & Hall, CRC Press, 2014.

[45] IBM. General Parallel File System. `http://www.ibm.com/systems/clusters/software/gpfs/index.html` (visited Jan 2015).

[46] IBM. *General Parallel File System Version 3 Release 5, Administration and Programming Reference*. International Business Machines Corporation, 2455 South Road, Poughkeepsie, NY 12601-5400, USA, document number sa23-2221-08 edition, 2013.

[47] Institute of Electrical IEEE and Inc. Electronics Engineers. *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX): System application program interface (API) C language, IEEE Std 1003.1*. IEEE Standards Office, New York, NY, USA, 2004-2013.

[48] Thomas Ilsche, Joseph Schuchart, Jason Cope, Dries Kimpe, Terry Jones, Andreas Knüpfer, Kamil Iskra, Robert Ross, Wolfgang E Nagel, and Stephen Poole. Enabling Event Tracing at Leadership-Class Scale through I/O Forwarding Middleware. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 49–60. ACM, 2012.

[49] Forschungszentrum Jülich Institute for Advanced Simulation, Theoretical Nanoelectronics. KKRnano: Korringa-Kohn-Rostoker Green function code for quantum description of nano-materials. `http://www.fz-juelich.de/ias/jsc/EN/Expertise/High-Q-Club/KKRnano/_node.html` (visited Jan 2015).

[50] IOR. IOR Parallel Filesystem test. `http://sourceforge.net/projects/ior-sio` (visited Jan 2015).

[51] Chen Jin, Scott Klasky, Stephen Hodson, Jay Lofstead, Fang Zheng, Matthew Wolf, and Robert Ross. *ADIOS User's Manual*. Oak Ridge National Laboratory, June 2013.

[52] Geir Johansen and Barb Mauzy. Cray XT Programming Environment's Implementation of Dynamic Shared Libraries. In *Proceedings of Cray User Group Meeting (CUG)*, Atlanta, Georgia, May 2009.

[53] Jülich Supercomputing Centre. JUQUEEN - Jülich Blue Gene/Q. `http://www.fz-juelich.de/ias/jsc/juqueen` (visited Sep 2015).

[54] Jülich Supercomputing Centre. JURECA - Jülich Research on Exascale Cluster Architectures. `http://www.fz-juelich.de/ias/jsc/jureca` (visited Sep 2015).

[55] Jülich Supercomputing Centre. JUROPA - Jülich Research on Petaflop Architectures. `http://www.fz-juelich.de/ias/jsc/juropa` (visited Sep 2015).

[56] Jülich Supercomputing Centre. JUST - Jülich Storage Cluster. `http://www.fz-juelich.de/ias/jsc/just` (visited Sep 2015).

[57] Suzanne M. Kelly, Ruth Klundt, and James H. Laros III. Shared Libraries on a Capability Class Computer. In *Proceedings of Cray User Group Meeting (CUG)*, Fairbanks, Alaska, May 2011.

[58] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir performance analysis tool-set. In *Tools for High Performance Computing*, pages 139–155. Springer, 2008.

[59] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, WolfgangE. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer Berlin Heidelberg, 2012.

[60] Quincey Koziol, Russ Rew, Mark Howison, Prabhat, and marc Poinot. *High Performance Parallel I/O*, chapter HDF5, pages 185–199. Computational Science Series. Chapman & Hall, CRC Press, 2014.

[61] Gregory L Lee, Dong H Ahn, Bronis R. de Supinski, John Gyllenhaal, and Patrick Miller. Pynamic: the Python Dynamic Benchmark. In *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pages 101–106. IEEE, 2007.

[62] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann/Academic, October 1999.

[63] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Robert Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 39–39. IEEE, 2003.

[64] Wei-keng Liao and Alok 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.

[65] Jay Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen 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.

[66] Sandra Loosemore, Richard M Stallman, Roland McGrath, Andrew Oram, and Ulrich Drepper. *The GNU C library reference manual.* Free software foundation, 2001.

[67] John M. May. *Parallel I/O for high performance computing*. Morgan Kaufmann, San Francisco, London, Sydney, 2001.

[68] Message Passing Interface Forum, Knoxville, Tennessee. *MPI: A Message-Passing Interface Standard*, 3.0 edition, September 2012.

[69] Message Passing Interface Forum, Knoxville, Tennessee. *MPI: A Message-Passing Interface Standard, Chapter 13 "I/O"*, 3.0 edition, September 2012.

[70] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst Simon. TOP500 supercomputer site. `http://www.top500.org` (visited Jan 2015).

[71] Patrick Miller. Parallel, Distributed Scripting with Python. In *Proceedings of the 3rd Linux Clusters Institute International Conference on Linux Cluster: The HPC Revolution*, Chatham, MA, USA, 2002. *IEEE* Computer Society Press, Los Alamitos, CA.

[72] Bernd Mohr, Brian J. N. Wylie, and Felix Wolf. Performance measurement and analysis tools for extremely scalable systems. *Concurrency and Computation: Practice and Experience*, 22(16):2212–2229, 2010. (ISC 2008 Award).

[73] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[74] MP2C: Massively Parallel Multi-Particle Collision Dynamics. `http://www.fz-juelich.de/ias/jsc/EN/Expertise/High-Q-Club/MP2C/_node.html` (visited Jan 2015).

[75] Oak Ridge National Laboratory. Titan Cray XK7 system. `https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/` (visited Jan 2015).

[76] Prabhat, Quincey Koziol, et al. *High Performance Parallel I/O*, chapter File Systems, pages 89–147. Computational Science Series. Chapman & Hall, CRC Press, 2014.

[77] Robert Ross, Gary Grider, Evan Felix, Mark Gary, Scott Klasky, Ron Oldfield, Galen Shipman, John Wu, et al. Storage Systems and Input/Output to Support Extreme Scale Science. Report of the DOE Workshops on Storage Systems and Input/output, Sponsored by the Office of Advanced Scientific Computing Research, U.S. Department of Energy, Rockville, Maryland, December 2014.

[78] rtld-audit man page. `http://man7.org/linux/man-pages/man7/rtld-audit.7.html` (visited Jan 2015).

[79] Kenichiro Sakai, Shinji Sumimoto, and Motoyoshi Kurokawa. High-performance and highly reliable file system for the K computer. *FUJITSU Science Technology*, 48(3):302–209, 2012.

[80] Frank Schmuck and Roger Haskin. GPFS: A Shared-disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST'02, Berkeley, CA, USA, 2002. USENIX Association.

[81] Till Schumann, Wolfgang Frings, Alexander Peyser, Wolfram Schenck, Kay Thust, and Jochen Martin Eppler. Modeling the I/O behavior of the NEST simulator using a proxy. In *Conference Proceedings of the YIC GACM 2015*. 3rd ECCOMAS Young Investigators Conference, Aachen (Germany), RWTH Aachen University, July 2015.

[82] Hongzhang Shan and John Shalf. Using IOR to analyze the I/O performance for HPC platforms. In *Proceedings of Cray User Group Meeting (CUG)*, Seattle, Washington, May 2007.

[83] Sameer S. Shende and Allen D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[84] Galen M. Shipman, David A. Dillow, Sarp Oral, Feiyi Wang, Douglas Fuller, Jason Hill, and Zhe Zhang. Lessons Learned in Deploying the Worlds Largest Scale Lustre File System. In *Proceedings of Cray User Group Meeting (CUG)*, May 2010.

[85] SIONlib: Scalable I/O library for parallel access to task-local files. `http://www.fz-juelich.de/jsc/sionlib/` (visited Oct 2015).

[86] Seung Woo Son, Saba Sehrish, Wei-keng Liao, Ron Oldfield, and Alok Choudhary. Dynamic File Striping and Data Layout Transformation on Parallel System with Fluctuating I/O Workload. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8, 2013.

[87] Spindle: Scalable loading of shared libraries of dynamically-linked HPC applications. `https://computation.llnl.gov/project/spindle/` (visited Oct 2015).

[88] Estela Suarez. DEEP-ER: Dynamical Exascale Entry Platform - Extended Reach, Software. `http://www.deep-er.eu/software` (visited Jan 2015).

[89] Godehard Sutmann and Wolfgang Frings. Extending scalability of MP2C to more than 250k compute core. In *Jülich Blue Gene/P Extreme Scaling Workshop 2009*, volume 2009 of *Technical Report FZJ-JSC-IB-2010-02*. Jülich Blue Gene/P Extreme Scaling Workshop, Jülich Supercomputing Centre, 2009.

[90] The Santa Cruz Operation and AT&T. System V Application Binary Interface, 1997. `http://refspecs.linuxbase.org/elf/gabi41.pdf` (visited Jan 2015).

[91] Unidata. Unidata's Network Common Data Form (netCDF). `http://www.unidata.ucar.edu/software/netcdf/` (visited Jan 2015).

[92] Interdisziplinäres Zentrum für Wissenschaftliches Rechnen Universität Heidelberg. $\mu\varphi$ (muPhi) a massive parallel program for the simulation of water flow and solute transport in porous media. `http://conan.iwr.uni-heidelberg.de/muphi/` (visited Jan 2015).

[93] Brian J. N. Wylie, David Böhme, Wolfgang Frings, Markus Geimer, Bernd Mohr, Zoltán Szebenyi, Daniel Becker, Marc-André Hermanns, and Felix Wolf. Scalable performance analysis of large-scale parallel applications on Cray XT systems with Scalasca. In *Proceedings of Cray User Group Meeting (CUG)*, Edinburgh, Scotland, May 2010. Cray User Group.

[94] Brian J. N. Wylie and Wolfgang Frings. Scalasca Support for MPI+OpenMP Parallel Applications on Large-scale HPC Systems Based on Intel Xeon Phi. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, XSEDE '13, pages 37:1–37:8, New York, NY, USA, 2013. ACM.

[95] Xyratex. Lustre file system. `http:lustre.org` (visited Jan 2015).

[96] Ilya Zhukov and Brian J. N. Wylie. Assessing Measurement and Analysis Performance and Scalability of Scalasca 2.0. In *Euro-Par 2013: Parallel Processing Workshops*, pages 627–636. Springer, 2014.

On current large-scale HPC systems often occur I/O patterns that produce a high load on the file system during access to checkpoint and restart files. Applications running on systems with distributed memory will often perform such I/O individually by creating task-local file objects on the file system. At large scale, these task-local I/O patterns impose substantial stress on the metadata management components of the I/O subsystem. Such metadata contention occurs also at the startup of dynamically linked applications while searching for library files.

The reason for these limitations is that the serial I/O components of the operating system do not take advantage of application parallelism. To avoid the above bottlenecks, this work describes two novel approaches which exploit the knowledge of application parallelism, the underlying I/O subsystem structure, the parallel file system configuration, and the network between HPC-system and I/O system to coordinate and optimize access to file-system objects. The underlying methods are implemented in two tools, SIONlib and Spindle, which add layers between the parallel application and the corresponding POSIX-based standard interfaces of the operating system, eliminating the need for modifying the underlying system software.

SIONlib is already applied in applications to implement efficient checkpointing and is also integrated in the performance-analysis tools Scalasca and Score-P to efficiently store trace data. Latest benchmarks on the Blue Gene/Q in Jülich demonstrate that SIONlib solves the metadata problem at large scale by running efficiently up to 1.8 million tasks while maintaining high I/O bandwidths of 60-80% of file-system peak with a negligible file-creation time. The scalability of Spindle could be demonstrated by running a benchmark on a cluster of Lawrence Livermore National Laboratory at large scale. The results show that the startup of dynamically linked applications is now feasible on more than 15000 tasks, whereas the overhead of Spindle is nearly constantly low.

With SIONlib and Spindle, this work demonstrates how scalability of operating system components can be improved without modifying them and without changing the I/O patterns of applications. In this way, SIONlib and Spindle represent prototype implementations of functionality needed by next-generation runtime systems.

IAS Series
Volume 30
ISBN 978-3-95806-152-1

JÜLICH
FORSCHUNGSZENTRUM