GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Research Training Report

Exploring the Performance of NetCDF Writing Routines in t8code

Jakob Hördt

MatrNr: 21565573

Supervisor: Jonathan Decker, Prof. Dr. Julian Kunkel

Georg-August-Universität Göttingen
Institute of Computer Science

1 March 2023

Abstract

t8code is an Adaptive Mesh Refinement (AMR) software based on Space-filling Curves (SFCs) for HPC applications that supports NetCDF as an output format. Use-cases for t8code include fluid dynamics simulations for airplanes and finite element methods, where it is applied to large problem sizes with high parallelism and produces large files. IO is often a bottleneck in HPC and impedes the scalability of applications, where t8code is no exception. In t8code, writing the output can take on the order of hours for typical use-cases even though modern NetCDF versions natively support parallel IO. Unfortunately, some precarious configuration options are left up to the NetCDF users. In this work, the performance of the NetCDF writing routines in t8code is explored via a series of benchmarks performed using a custom-made, configurable benchmark suite called t8cdfmark. This work contributes an implementation of file-per-process IO to t8code and shows that it outperforms NetCDF parallel IO by up to a factor of five. Another speedup is possible by using Earth-System Data Middleware (ESDM), which is a comprehensive parallel IO middleware, a part of which allows using it as an alternative implementation of the NetCDF interface. It also offers a speedup of up to five over plain NetCDF IO. The conducted benchmarks confirm that t8code chose good default NetCDF parameters, but also show that the collective and independent IO settings in combination with contiguous storage yield up to 1.5 speedup over the respective other setting when adjusted for parallelism and problem size.

Acknowledgements

Thanks to Jonathan Decker and Prof. Dr. Julian Kunkel from the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG) for advising me and providing invaluable feedback during this work. Thanks to Pavan Kumar Siligam from the GWDG for helping me with the Earth-System Data Middleware (ESDM) usage and installation. Furthermore, I want to thank Dr. Johannes Holke and Niklas Böing from the Deutsches Zentrum für Luft- und Raumfahrt e. V. (DLR) for their support regarding t8code and for being open to my contributions.

Contents

List of Figures

List of Listings

List of Abbreviations

AMR Adaptive Mesh Refinement

API Application Programming Interface

CLI Command Line Interface

CPU Central Processing Unit

CSV Comma-separated values

DLR Deutsches Zentrum für Luft- und Raumfahrt e. V.

ESDM Earth-System Data Middleware

GCC GNU Compiler Collection

GNU GNU's Not Unix

GWDG Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

HDF Hierarchical Data Format

HPC High-Performance Computing

IO Input / Output

JSON JavaScript Object Notation

MPI Message Passing Interface

NetCDF Network Common Data Form

OGC Open Geospatial Consortium

POSIX Portable Operating System Interface

RAM Random Access Memory

S3 Simple Storage Service

SCC Scientific Compute Cluster

SFC Space-filling Curve

TM-SFC Tetrahedral Morton Space-filling Curve

1 Introduction

"t8code[1] (spoken as "tetcode") is a C/C++ library to manage parallel adaptive meshes with various element types[...]" [Hol+22], which is suitable for use in High-Performance Computing (HPC) applications like numeric simulations. The adaptive meshes are managed in a forest, which is a hierarchical data structure. t8code recently gained support for exporting these forests as a Network Common Data Form (NetCDF)[2] file[3]. NetCDF is a popular self-describing scientific file format that supports large datasets [22].

According to Behzad et al."scalability of applications is often limited by poorly performing parallel I/O" and "I/O can be a significant bottleneck on HPC application performance." [Beh+19]. The costs incurred by this in terms of execution time, energy, and, therefore, money are amplified by the scale of HPC systems, making performance and file Input / Output (IO) a larger concern than in non-HPC applications.

1.1 Goals

The goal of this research training is to shed light on specifically the t8code NetCDF IO performance by:

- Collecting benchmark data to support any efforts in optimizing the t8code NetCDF writing performance.
- Creating a benchmark suite that models common use-cases for t8code and NetCDF to allow the reproducible creation of benchmark data. The suite should be open to anyone as it may be useful to others in their optimization efforts. This will be achieved by making the repository public under a permissive license.
- Using the created benchmark data to create actionable guidelines for t8code use depending on use-case arguments and ideally make concrete improvements to the writing implementation in t8code or provide more options, like file-per-process mode, if they prove a significant advantage in some use-cases.

1.2 Contributions

To achieve these goals I developed a benchmark suite to perform comprehensive benchmarks on the NetCDF writing functionality and contributed a pull request to the t8code project that extends a NetCDF routine to allow for more customization of performance related parameters. In addition, I implement a different writing mode that lets each process write only its process local data into its own NetCDF file. This is commonly referred to as file-per-process IO. I also evaluate the performance of using ESDM as a NetCDF library replacement. ESDM is introduced in section 2.3.

[1] t8code source, *Accessed on 10 November 2022*: `https://github.com/DLR-AMR/t8code`

[2] NetCDF, *Accessed on 10 November 2022*: `https://www.unidata.ucar.edu/software/netcdf/`

[3] t8code NetCDF pull-request, *Accessed on 10 November 2022*: `https://github.com/DLR-AMR/t8code/pull/191`

1.3 Outline

Firstly, this report covers technical background information in section 2. In section 3, the changes made to t8code, including the benchmark program I added, are explained. Next, the benchmark methodology and approach is explained in section 4. The specification for the t8cdfmark benchmark suite is found in section 5. Lastly, I will present and discuss the benchmark results in section 6 and section 7, and draw conclusions and give an outlook on future work in section 8.

2 Background

2.1 t8code

Dr. Johannes Holke developed t8code [Hol+22] as part of his dissertation on "Scalable Algorithms for Parallel Tree-based Adaptive Mesh Refinement with General Element Types" [Joh18].

Adaptive Mesh Refinement (AMR) is a common optimization in simulations that aims to refine simulation space where it is required to keep up the numeric simulation accuracy, or colloquially, where a lot is happening, and coarsen simulation space, where doing so does not relinquish too much accuracy, to save computational power. In t8code, simulation space is modeled by a so called coarse mesh, which is a collection of primitives. A primitive is one of the atomic shapes t8code supports, like hexahedra, tetrahedra, prisms, and pyramids. A coarse mesh can be turned into a forest, where its primitives become trees. The primitives are referred to as trees, because each acts as a root for AMR. An element is a primitive that is a leaf node of a tree. When an element is refined, the finer elements are the children of the original parent element. The term forest stems from it being a collection of trees.

As Dr. Johannes Holke explains in this work, for representing a tree in t8code, a Space-filling Curve (SFC) is used. A space-filling curve index is a function from the refinement space into the natural numbers with some additional properties. For example, the indices of descendants of an element are between the index of the ancestor element and any element with a higher index. This means that refining an element does not change indices outside of the element, making it a local operation. The index imposes an ordering on the multi-dimensional elements that allows efficient storage in an array format. In his work the author also develops efficient algorithms for finding the children, parent, or face-neighbors of simplex elements in this SFC based representation. Furthermore, the author shows that SFC based indices can be extended to order the elements of multiple trees. This is visualized in fig. 1.

One of the key innovations explained in the dissertation is to develop a novel SFC index, the Tetrahedral Morton Space-filling Curve (TM-SFC), which also encodes the primitive type of an element. This allows the t8code library to be generic over various element types, like hexahedra, tetrahedra, prisms, and pyramids, and use mixed element types in the same mesh. This is called a hybrid mesh.

Since t8code aims to tackle large problem sizes, it is designed to run on HPC systems. HPC systems typically consist of a multitude of computers connected via a network interconnect. In such a setting, shared memory parallelism is difficult to implement since fast memory shared by all computers is not naturally available. A distributed mem-

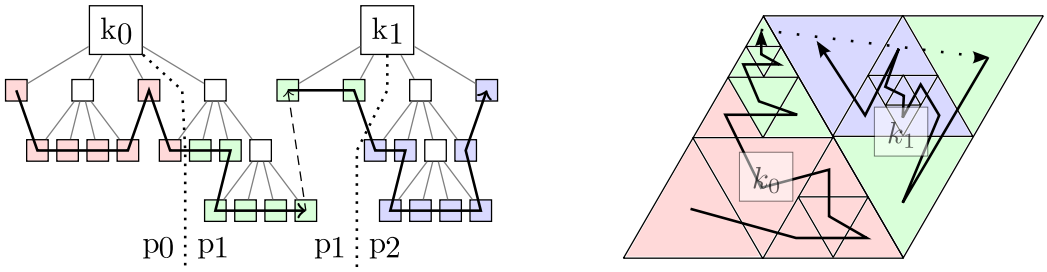

Figure 1: From [Joh18, Figure 3.5]. This figure shows a forest of two trees k_0 and k_1. On the right, the corresponding mesh is displayed. On the left, the corresponding tree data structure is visualized. The elements, shown as boxes on the left and as triangles on the right, are distributed into equal parts among the processes p_0, p_1, and p_2. The color indicates the process number. The figure also shows how an SFC index can be extended to multiple trees by simply connecting multiple SFCs according to a predetermined order.

ory paradigm like message passing is better suited in this case. Message Passing Interface (MPI) is such a standardized Application Programming Interface (API) specification that enables process communication through message passing[4]. t8code uses MPI for its parallelism. Note that MPI implementations *can* use shared memory for faster communication if available, for example when two MPI processes run on the same Central Processing Unit (CPU).

To achieve parallelism on a cluster, it is necessary to distribute the workload across the computers, which is called load balancing or (re-)partitioning in t8code's terms. During simulation, a mesh is often dynamically adapted, which may lead to unbalanced tasks by changing the number of elements each process owns. The SFC based ordering enables repartitioning the elements in linear time by splitting the ordered element list into roughly equally sized chunks.

2.2 NetCDF

NetCDF is one of the scientific file formats that t8code can export to, which is the step that is benchmarked in this work. NetCDF is an international standard by the Open Geospatial Consortium (OGC). The NetCDF format is designed to be self-describing, meaning that machine readable metadata describing the semantics and structure of the data is part of the file, making the data portable and accessible. A NetCDF file can always be opened by means of the standard NetCDF APIs. NetCDF is also platform independent, meaning NetCDF files written on one platform can be accessed on any other.

The NetCDF specification and libraries have been continuously developed and today several supported versions of the NetCDF format exist. Three of those versions are considered classic. The default format, also called CDF-1, the CDF-2 format enabled by the `NC_64BIT_OFFSET` flag during creation, and the CDF-5 format enabled by the `NC_64BIT_DATA` flag. The newest supported format is called netCDF-4/HDF5. This is the format used by t8code when I started my work. As the name implies, netCDF-4/HDF5 is built on top of the Hierarchical Data Format (HDF)5 format[5]. A netCDF-4/HDF5 file is always a valid HDF5 file. HDF5, similar to NetCDF, is a file format to store large amounts of data

[4]MPI home page, *Accessed on 10 November 2022*: `https://www.mpi-forum.org/`
[5]HDF5 website, *Accessed on 10 November 2022*: `https://www.hdfgroup.org/solutions/hdf5/`

in a portable and self-describing fashion. NetCDF supports only restricted functionality compared to HDF5. Non hierarchical group structures and references for example are disallowed in NetCDF[6]. NetCDF strives to hide HDF5's complexity behind its APIs[7].

Every NetCDF file has a header containing metadata as well as a data part containing the actual data[8]. Notably, the variables and dimensions of the file are declared in the header. Each variable is a multidimensional array of only one type like double, integer, or a custom one. A dimension in NetCDF is simply a named integer that can be used as dimensions for variables. This data model is flexible and translates directly to many use-cases, like for example data sets on a uniform three-dimensional grid. For t8code, whose data model is an adaptively refined mesh, the NetCDF data model does not translate directly. To represent the t8code mesh in NetCDF's data model, t8code uses the UGRID Conventions[9]. Specifically, for two and three dimensional data, the conventions for a 2D flexible mesh, and for a fully 3D unstructured mesh topology are used respectively. Data of different dimensionality cannot be exported from t8code as NetCDF. In these conventions, a mesh consists of volumes, which are a set of connected nodes. Nodes are simple vertices with a single set of coordinates. In NetCDF, volumes are mainly described by a variable called `Mesh3D_vol_nodes`, which, for each volume, lists the indices of the associated nodes. The nodes are described by multiple variables, one for each coordinate dimension.

NetCDF supports parallel IO on all four formats listed above. For the classic formats parallel IO is supported by the PnetCDF[10] library, which was originally an independently developed NetCDF interface [Li+03] and is now part of NetCDF. Since netCDF-4/HDF5 files are based on HDF5, parallel IO on this format is supported by parallel HDF5 implementations[11]. Both PnetCDF and parallel HDF5 are implemented using MPI-IO, a part of MPI-2 and later, that specifies parallel file IO[Mes09]. To create or open a file in parallel, NetCDF offers the `nc_create_par` and `nc_open_par` functions respectively. In addition, the `nc_var_par_access` function is used to specify the access mode. When using HDF5, the function applies to each NetCDF variable individually, while, when using PnetCDF, the function always applies to all variables.[12]. The access mode can be either collective or independent. In collective mode, all MPI processes wait for each other in the call and then perform the access together in a coordinated fashion. This introduces synchronisation overhead but enables certain optimizations like combining IO operations to reduce the amount of IO calls. In independent mode, processes do not need to cooperate or wait for others, reducing synchronisation overhead over collective mode but preventing other optimizations. NetCDF generally recommends collective mode[13]. Another way to

[6]Interoperability with HDF5, *Accessed on 14 November 2022*: `https://docs.unidata.ucar.edu/netcdf-c/current/interoperability_hdf5.html`

[7]What NetCDF users should know about HDF5?, *Accessed on 14 November 2022*: `https://www.unidata.ucar.edu/software/netcdf/workshops/2007/hdf5/ncw07-hdf5.pdf`

[8]The Components of a NetCDF Data Set, *Accessed on 26 July 2022*: `https://docs.unidata.ucar.edu/nug/current/netcdf_data_set_components.html`

[9]UGRID Conventions (v1.0), *Accessed on 31 August 2022*: `https://ugrid-conventions.github.io/ugrid-conventions/`

[10]PnetCDF website, *Accessed on 10 November 2022*: `https://parallel-netcdf.github.io/`

[11]Parallel HDF5, *Accessed on 10 November 2022*: `https://www.hdfgroup.org/2015/08/parallel-io-with-hdf5/`

[12]nc_var_par_access docs, *Accessed on 10 November 2022*: `https://docs.unidata.ucar.edu/netcdf-c/current/group__datasets.html#ga6dc46e4ab82584360518db5cb0cad841`

[13]NetCDF collective mode, *Accessed on 10 November 2022*: `https://docs.unidata.ucar.edu/netcdf-c/current/group__datasets.html#ga6dc46e4ab82584360518db5cb0cad841`

parallelize file IO that does not rely on MPI-IO is described in section 3.1.

For the described parallel IO facilities to yield any speedup over serial IO, the underlying file system must also support parallel IO. One such file system is briefly described in section 4.1.

The t8code interface to write NetCDF files consists of the two functions `t8_forest_write_netcdf` and `t8_forest_write_netcdf_ext`. `t8_forest_write_netcdf` has less paramters, is implemented in terms of `t8_forest_write_netcdf_ext`, and specifies default arguments. `t8_forest_write_netcdf_ext` is supposed to be an advanced function. Both take a forest, a file name, a title, and optionally, element-wise data variables. The `*_ext` function additionally takes the MPI access mode described above, and the storage mode, also known as data layout. NetCDF can write in contiguous, or in chunked mode. In contiguous mode the data is written contigously, while in chunked mode, the data is written in hyperslab shaped chunks[14]. The contributed parameter changes to `t8_forest_write_netcdf_ext` are described in section 3.

2.3 ESDM

ESDM[15] is a comprehensive software system for improving IO performance in HPC workflows. It achieves this by:

- Utilizing structural information exposed by data description formats like HDF5 and NetCDF.[16]
- Enabling the transparent use of heterogeneous storage architectures. It is common for HPC systems to have multiple storage partitions with different characteristics. In a typical example, a fast scratch partition with low failure protection and regular cleanup, a persistent but slower home or work partition, and very fast node local storage is available. Additionally, non filesystem storage technologies such as Seagate Motr or Amazon Simple Storage Service (S3) may want to be used. ESDM provides backends for the above, and more, storage targets. File systems are supported by a Portable Operating System Interface (POSIX) based backend. ESDM consumes a configuration file describing the aforementioned characteristics. This information is used to prioritise IO targets and distribute data efficiently across them[KP20, Section 4.1]
- Utilizing workflow information. Users of ESDM may provide a workflow file specifying input and output datasets and corresponding information like lifetime, frequency of the output, or expected size of the datasets. ESDM can use this information to automatically decide where to put datasets and optimize workflow schedules.[KP20, Section 4.1-4.3]

ESDM provides a library enabling it to act as a replacement for the NetCDF C library. Since t8code uses NetCDF, it is possible to use t8code with ESDM as well. In this work, the performance of this approach will be evaluated with the POSIX ESDM backend. In

[14]information on chunking, *Accessed on 10 November 2022*: `https://docs.unidata.ucar.edu/nug/current/netcdf_perf_chunking.html`

[15]ESDM, *Accessed on 17 January 2023*: `https://hps.vi4io.org/products/esdm`

[16]ESDM may use HDF5 and NetCDF information, *Accessed on 29 January 2023*: `https://github.com/ESiWACE/esdm/blob/e49a9d9ed08f25a6f31eac9b2c632a262282f91a/README.md#earth-system-data-middleware`

ESDM, data sets are split into so called fragments, which hold a continuous sub-domain of the data and are stored on one storage target each[17]. Of the aforementioned ESDM features and performance advantages, only the performance of the splitting into fragments and their mapping into the POSIX back end are considered here.

3 t8code Contribution

As stated in section 1, as part of this research training, I contributed a pull-request to the t8code repository[18]. The pull-request includes the implementation of file-per-process mode described in section 3.1, as well as a Command Line Interface (CLI) program, elaborated on in section 3.2, to benchmark various writing configurations without re-compilation. Furthermore, the pull-request extends the `t8_forest_write_netcdf_ext` function to allow customizing more parameters.

Namely, the creation mode parameter of the NetCDF `nc_create`[19] and `nc_create_par` functions can now be customized. Only `NC_CLOBBER` is added to the creation mode before it is passed to `nc_create(_par)`. Supported arguments are `NC_NETCDF4`, which creates a netCDF4/HDF5 file, or `NC_64BIT_DATA`, which creates a CDF5 file. Other versions are non trivial to support because t8code uses 64 bit integers, denoted by `NC_INT64` in NetCDF, for example to represent node indices, which are not available in earlier NetCDF versions. Therefore, other versions are not implemented in the pull-request. CDF5 was implemented because there was interest in the performance of a NetCDF format not backed by HDF5.

The fill mode in NetCDF can be `NC_FILL` or `NC_NOFILL`. It can also be customized using `t8_forest_write_netcdf_ext` now. With fill turned on NetCDF prefills all variables with so called fill values to indicate that no value is present. This allows NetCDF to detect accesses to data that has not yet been written. For creating a fresh NetCDF dataset, it is recommended to disable fill mode to avoid duplicate writes to increase performance[20]. This was contributed because it can help debugging writing NetCDF files.

Another contribution is that in chunked storage mode, the chunksize for the, depending on the dimension, two or three coordinate variables can be specified, which may improve performance. For the other variables, and for the coordinate variables, if the given chunksize is `nullptr`, the NetCDF default chunksizes are used in chunked mode.

The difference between the signatures of `t8_forest_write_netcdf_ext` before and after the contribution is shown in listing 1.

[17] Progress of WP4: Data at Scale, Fragment definition, *Accessed on 29 January 2023*: `https://hps.vi4io.org/_media/research/talks/2020/2020-05-27-progress_of_wp4_data_at_scale.pdf`

[18] Contribution to t8code, *Accessed on 10 November 2022*: `https://github.com/DLR-AMR/t8code/pull/279`

[19] nc_create docs, *Accessed on 10 November 2022*: `https://docs.unidata.ucar.edu/netcdf-c/current/group__datasets.html#ga427f5a0b24f1d426a99bcc37b8a39cac`

[20] `nc_set_fill` docs, *Accessed on 10 November 2022*: `https://docs.unidata.ucar.edu/netcdf-c/current/group__datasets.html#ga610e6fadb14a51f294b322a1b8ac1bec`

```
 void t8_forest_write_netcdf_ext (t8_forest_t forest,
         const char *file_prefix,
         const char *file_title,
         int dim,
         int num_extern_netcdf_vars,
         t8_netcdf_variable_t *
         ext_variables[],
         sc_MPI_Comm comm,
         int netcdf_var_storage_mode,
-        int netcdf_var_mpi_access);
+        const size_t
+        *coordinate_chunksize,
+        int netcdf_var_mpi_access,
+        int fill_mode, int cmode,
+        bool file_per_process_mode);
```

Listing 1: Contribution to t8_forest_write_netcdf_ext

3.1 file-per-process mode

Historically, parallel IO has been achieved with the aptly named file-per-process model. In this model, each parallel process accesses a different file[21]. The parallel execution of all the independent IO requests is left completely to the parallel filesystem. The advantage of this approach is that it is conceptually simpler than single-file parallel IO and easy to implement on the user side. A parallel IO library like MPI-IO is not required. Furthermore, the approach promises good performance, due to requiring little synchronisation overall. Unfortunately, in reality file-per-process often suffers from file system contention on larger process counts. Another drawback is that you end up with as many files as processes which cannot be read as easily as a single file.

Nevertheless, to compare its performance characteristics to the parallel IO facilities built into NetCDF, I implemented file-per-process as an optional writing mode. This is done by modifying the `t8_forest_write_netcdf_ext` function to accept an additional boolean parameter that indicates whether file-per-process mode is desired. If that is the case, the function generates a filename unique to each process and creates a NetCDF file at the resulting path using `nc_create` instead of `nc_create_par`. That way, each process can write to its own, independent, serial NetCDF file. In single file mode, each process calculates the offset at which it writes to the shared file, for each variable. This is set to zero in file-per-process mode. Furthermore, in file-per-process mode, the MPI rank and the MPI size are written to each process' files in the form of global NetCDF attributes. This is all the information needed to attach the files back together correctly. The function branches in a few more places, for example to prevent `nc_var_par_access` from being called in file-per-process mode.

[21] file-per-process, slide 14, *Accessed on 10 November 2022*: `https://www.olcf.ornl.gov/wp-content/uploads/2011/10/Fall_IO.pdf`

```
srun ./t8_write_forest_netcdf 100000000000 NC_NOFILL netcdf4_hdf5
↪  NC_CONTIGUOUS NC_COLLECTIVE
```

Listing 2: Example usage of the CLI contributed to t8code

3.2 Benchmark CLI

The CLI program contributed to t8code creates a single forest and times the export of the forest as NetCDF. This program forms the basis for the main CLI that is now part of the t8cdfmark project specified in section 5. The various parameters of the NetCDF writing functionality can be customized via the CLI. For example you can specify that the export should be done in file-per-process mode. The CLI has a reduced feature set compared to the one from t8cdfmark. Namely, neither does this program allow creating element-wise data variables, nor are different forest models supported. This CLI uses only the pseudo random model described in section 5.1. This allows the program to take a hint for the written storage. The forest is created in a way that it roughly occupies the given storage. For details on how this is achieved consult section 5.1. The example shown in listing 2 creates a NetCDF file roughly 100GB in size. A small manual for the CLI is part of the pull-request to t8code.

4 Methodology

As discussed in section 1, one of the goals of this work is to produce benchmark data and draw conclusions and recommendations from them. One of the general questions this work attempts to answer is how the different writing configurations scale with the number of nodes or the number of processes per node. Unfortunately, the many combinations of arguments, like node counts, processes per node, written storage, number of element-wise variables, and the arguments to `t8_forest_write_netcdf_ext` make it infeasible to perform a complete grid search over the large parameter space. Instead the benchmarks need to be focused on specific questions and a few arguments at a time.

Here, the search space is reduced by focusing on a strong scaling series and a weak scaling series. During strong scaling, the parallelism is increased on a fixed problem size. Ideally, the time to complete the task should decrease proportional to the parallelism. During weak scaling, the parallelism is increased proportionally in lockstep with the problem size. Ideally, the time should stay constant. The benchmark results presented here stray far from those ideals but pursuing them is not the main goal here. In traditional scaling tests, a common question is how much parallelism to allocate for a program for a given problem size. However, since this work inspects only the writing performance of t8code and not the program as a whole, the goal is instead to answer the question of how to maximize the throughput with the given parallelism. In practice the allocated parallelism would be estimated with an analysis on t8code as a whole. With this in mind I decided against showing speedup diagrams and instead to just plot the throughput over the parallelism in logarithmic scale. This is a better visualization for answering the posed questions. Nonetheless, strong and weak scaling sets of benchmarks allow us to judge a configurations performance not only by its absolute timing, but also by its behaviour when scaling up.

The strong scaling test is further split up by increasing the parallelism through the

tasks per node and the node count, while the respective other is fixed. The tasks per node and the node count are scaled up independently firstly to see the effects of them separately. A possible finding could have been that it makes sense in certain configurations to run t8code on more nodes with less tasks per node than available to get more storage bandwidth without increasing the total process count. Secondly, scaling the two parameters independently was done in an attempt to make the resulting data easier to interpret. Scaling up just the total tasks naively would result in unaccounted jumps in the node count to fulfill the task requirement, complicating the analysis.

In the weak scaling series, the tasks per node count is fixed and the node count is scaled up. Scaling up only the tasks per node and problem size would not be a realistic scenario for a weakly scaling application and is therefore not considered. In this experiment, the problem size grows from roughly 15GB on one node to writing roughly 1TB on 64 nodes. Five tasks per node was chosen to be able to test up to 64 nodes while using appropriate parallelism for the problem sizes and still keeping the benchmark runtime manageable.

These three benchmark series are repeated with all possibilities for the storage mode, the MPI access mode, and the NetCDF version. Notably, the contributed file-per-process mode described in section 3.1 is among the storage modes. All of this is repeated with ESDM to assess its performance in this use-case.

The benchmarks are facilitated by the custom-made t8cdfmark benchmark suite specified in section 5, which includes a CLI that executes a single writing benchmark with the given arguments. This is described in detail in section 5.1. It derives from the CLI contributed to t8code, see section 3.2, but also allows supports customizing the element-wise variables to a more realistic value. A script is used to schedule multiple runs of the benchmark CLI with different arguments and also allows repeating the same benchmarks. Here, all benchmarks are repeated three times to gain some statistical confidence. This facility is explained in section 5.2. The resulting data is collected in a separate step using another tool that is part of t8cdfmark, see section 5.3.

4.1 SCC

The benchmarks are executed on the Scientific Compute Cluster (SCC)[22], which is a modern HPC cluster operated by the GWDG[23]. The SCC has a shared filesystem called scratch, where the files for our benchmarks are written. For our purposes, at most 96 nodes have access to it. The filesystem powering scratch is BeeGFS[24], which is a clustered parallel filesystem, meaning it can be mounted and run on multiple servers. The metadata and data servers are separate components in BeeGFS allowing them to be scaled independently onto many computers[25]. At the GWDG, scratch is backed by two server blades, each with two metadata and two data service instances. The approximately 220 storage drives are distributed across three separate external enclosures. Theoretically, all of them can write data simultaneously. For job scheduling, the SCC uses the Slurm[26] scheduler.

[22]SCC, *Accessed on 10 November 2022*: `https://www.gwdg.de/web/guest/hpc-on-campus/scc`

[23]GWDG website, *Accessed on 15 February 2023*: `https://www.gwdg.de/`

[24]BeeGFS, *Accessed on 1 September 2022*: `https://www.beegfs.io/c/`

[25]How BeeGFS Works, *Accessed on 1 September 2022*: `https://www.beegfs.io/c/home/how-beegfs-works/`

[26]Slurm, *Accessed on 10 November 2022*: `https://slurm.schedmd.com/`

4.2 Benchmark configurations

For the benchmarks, the t8cdfmark CLI from commit b5cea7c[27] was compiled with Open MPI version 4.1.1[28] and GNU Compiler Collection (GCC) version 9.3.0[29] and run with netcdf-c version 4.8.1[30] and HDF5 version 1.10.7[31].

In real world simulations, between one and hundreds of variables per element are used. Since this value is not expected to have a big performance impact by itself, all benchmarks use ten variables per element. For the strong scaling benchmarks, the size hint is fixed at 100 GB resulting in about 85 GB of actually written data. Again, this is a plausible value for research with t8code. In practice, bigger datasets appear, but this is big enough to compare throughputs on up to 64 nodes. In the weak scaling series, the problem size is grown from 13 GB to 845 GB, which is a relatively big, but still realistic problem size.

```
def configurations():
  for repetition in range(3):
    for comm_mode in ["NC_INDEPENDENT", "NC_COLLECTIVE",
    ↪  "file_per_process"]:
      for storage_mode in ["NC_CONTIGUOUS", "NC_CHUNKED"]:
        for cmode in ["netcdf4_hdf5", "cdf5"]:
          if cmode == "cdf5" and comm_mode != "file_per_process":
            continue

          for nodes in [1,4,16,64]:
            yield Config(repetition=repetition, nodes=nodes,
            ↪  tasks_per_node=5, storage_mode=storage_mode, cmode=cmode,
            ↪  comm_mode=comm_mode, num_element_wise_variables=10,
            ↪  bytes_hint=100000000000)
          for tasks_per_node in [1,10,20]:
            yield Config(repetition=repetition, nodes=16,
            ↪  tasks_per_node=tasks_per_node, storage_mode=storage_mode,
            ↪  cmode=cmode, comm_mode=comm_mode,
            ↪  num_element_wise_variables=10, bytes_hint=100000000000)
```

Listing 3: Configuration for strong scaling benchmarks

The exact config files ingested by the t8cdfmark tools described in section 5.2 and section 5.3 used for strong and weak scaling are shown in listing 3 and listing 4 respectively. For the benchmarks using ESDM, a separately compiled t8cdfmark binary with ESDM support is used. The used ESDM config file is suitable for the SCC and is part of the supplement in section 8.

[27] Commit b5cea7c of t8cdfmark, *Accessed on 17 January 2023*: https://github.com/neoq/t8cdfmark/tree/b5cea7caaa305c8cf154ced6d0cc6c2d435eb949

[28] Open MPI version 4.1, *Accessed on 17 January 2023*: https://www.open-mpi.org/software/ompi/v4.1/

[29] GCC 9, *Accessed on 17 January 2023*: https://www.gnu.org/software/gcc/gcc-9/

[30] netcdf-c version 4.8.1, *Accessed on 17 January 2023*: https://github.com/Unidata/netcdf-c/releases/tag/v4.8.1

[31] HDF5 version 1.10.7, *Accessed on 17 January 2023*: https://www.hdfgroup.org/2020/09/release-of-hdf5-1-10-7-newsletter-175/

```
def configurations():
  for repetition in range(3):
    for comm_mode in ["NC_INDEPENDENT", "NC_COLLECTIVE",
    ↪  "file_per_process"]:
      for storage_mode in ["NC_CONTIGUOUS", "NC_CHUNKED"]:
        for cmode in ["netcdf4_hdf5", "cdf5"]:
          if cmode == "cdf5" and comm_mode != "file_per_process":
            continue
          for nodes in [1,4,16,64]:
            yield Config(repetition=repetition, nodes=nodes,
            ↪  tasks_per_node=5, storage_mode=storage_mode, cmode=cmode,
            ↪  comm_mode=comm_mode, num_element_wise_variables=10,
            ↪  bytes_hint=1000000000000//64*nodes)
```

Listing 4: Configuration for weak scaling benchmarks

5 t8cdfmark

As part of this research training I designed an additional benchmark suite that is not part of t8code but rather depends on it. In this section, t8cdfmark will be specified for documentation purposes but also to guide the implementation. The goal of t8cdfmark is to model real-world use cases with t8code and benchmark the NetCDF writing functionality. Benchmarking t8code functionality besides NetCDF writing is an explicit non-goal. Only the time the writing takes will be measured. The following use-cases are supported by the design:

- As a t8code user, I want to use t8cdfmark to find writing parameters that suit my use-case well, by picking a t8cdfmark scenario that resembles my use-case.
- As a researcher, I want to use t8cdfmark to evaluate writing parameters to give recommendations to t8code users and improve t8code defaults.
- As a researcher, I want to use t8cdfmark to evaluate modifications to the t8code writing routines for performance.

Nevertheless, t8cdfmark is not designed to be used as a benchmark for the underlying hardware. t8cdfmark includes an executable that benchmarks a single configuration, as opposed to multiple, which is given via a CLI. Furthermore, t8cdfmark includes Python scripts for launching t8cdfmark with many combinations of arguments and accumulating the results in human, and machine readable form.

5.1 CLI specification

The t8cdfmark CLI is an MPI program that can be run in a user controlled MPI configuration. To benchmark a set of arguments, the CLI first constructs a forest by creating a coarse mesh and refining it according to the selected model, all of which are described below, and then partitioning it among the processes. The resulting forest is afterwards exported in NetCDF format using the functionality provided by t8code. This part is timed

using two calls to `MPI_Wtime` before and after the operation respectively. The CLI offers various models, which are different initial forests with a refinement scheme, to export. The following models are supported:

- uniform, where every element is at the same level of refinement.
- pseudo-random, where elements of a uniformly refined forest are refined at most one level according to a pseudo-random number generator.
- towers-of-hanoi shaped, where a uniformly refined forest is refined one level inside of the specified radius. If more radii are given, under the condition that they are monotonically decreasing, the forest is further refined one level in each of the given radii.
- ring shaped, where a forest is refined strongly in proximity to the border of a sphere, but not on the inside.
- A mesh that models a plane that is refined close to the border.

When choosing the uniform, or pseudo random models, the CLI accepts a storage size hint parameter. The hint is used to calculate refinement arguments so that the resulting forest consumes roughly the given size on disk. This is useful, because often, when benchmarking storage, there is interest in specifying a rough benchmark size. The basis for both models is a cube shaped coarse mesh created with `t8_cmesh_new_hypercube_hybrid`. The cube consists of six tetrahedra, six triangular prisms, and four hexahedra[32] for a total of sixteen trees or initial elements without refinement.

Each of those primitive, when refined, is replaced by eight finer elements[33]. The storage requirement for a single element in NetCDF form can be estimated upwards using eq. (7), where num_element_wise_variables represents the number of element-wise data variables, each assumed to occupy 8 B per element. With nMaxMesh3D_vol_nodes = 8 and nMesh3D_node ≤ nMesh3D_vol·nMaxMesh3D_vol_nodes, because the exact node count is unknown in this estimate, we get:

$$\text{storage} \leq \text{nMesh3D\_vol} \cdot (4 + 8 + 64 + 196 + \text{num\_element\_wise\_variables} \cdot 8) \quad (1)$$

$$\frac{\text{storage}}{\text{nMesh3D\_vol}} = \text{bytes\_per\_element} \leq 268 + \text{num\_element\_wise\_variables} \cdot 8 \quad (2)$$

$$(3)$$

The number of elements to occupy the given storage is now estimated by:

$$\text{nMesh3D\_vol} \geq \text{storage} \div (268 + \text{num\_element\_wise\_variables} \cdot 8) \quad (4)$$

Using this fact, the CLI calculates a lower bound for the number of elements that the forest can have without occupying more storage in NetCDF form than the given size. The CLI then calculates an initial refinement, which is applied uniformly to the aforementioned coarse mesh by means of `t8_forest_new_uniform`, as well as a ratio

[32] `t8_cmesh_new_hypercube_hybrid`, *Accessed on 10 November 2022*: `https://github.com/DLR-AMR/t8code/blob/c28db8862b0877b3f1d407a7d7263343c89a08d6/src/t8_cmesh/t8_cmesh_examples.h#L104`

[33] refinement of different element types, *Accessed on 10 November 2022*: `https://github.com/DLR-AMR/t8code/blob/c28db8862b0877b3f1d407a7d7263343c89a08d6/README.md`

of additionally refined elements. Let i be the initial refinement, and a the additionally refined ratio. Then

$$\text{nMesh3D\_vol} = (1-a)\,16 \cdot 8^i + a \cdot 16 \cdot 8^{(i+1)} \tag{5}$$

$$\Rightarrow \begin{cases} i = \text{initial refinement} = \lfloor \log\_8(\text{nMesh3D\_vol}/16) \rfloor \\ a = \text{additional refinement ratio} = \dfrac{\text{nMesh3D\_vol}}{7 \cdot 16 \cdot 8^i} - \dfrac{1}{7} \end{cases} \tag{6}$$

In pseudo random refinement mode, the CLI uses a pseudo random number generator with a Bernoulli distribution to conditionally refine elements to get approximately the previously calculated number of elements. In the uniform scenario, the additional adaption step is not done. The other scenarios do not take a storage hint and the written forest always has the same storage requirement.

As discussed above, elements can have less than eight nodes, reducing the storage size of the forest. For this reason, the CLI also calculates the actual storage size, after the forest is created, in every scenario, with the true number of vertices, according to eq. (7), which is derived from the UGRID conventions[34].

$$\begin{aligned} \text{actual storage} = {} & \text{nMesh3D\_vol} \cdot 4 + \text{nMesh3D\_vol} \cdot 8 + \\ & \text{nMesh3D\_vol} \cdot \text{nMaxMesh3D\_vol\_nodes} \cdot 8 + 3 \cdot \text{nMesh3D\_node} \cdot 8 + \\ & \text{num\_element\_wise\_variables} \cdot \text{nMesh3D\_node} \cdot 8 \end{aligned} \tag{7}$$

The CLI prints this result immediately to give the user the opportunity to cancel the benchmark, if the size is undesirable, for example, too small. The actual size is also part of the output, as shown in listing 5, and is used for the throughput calculation. The actual size is used because file-per-process mode has a certain storage overhead per file which is not useful information and should not be counted towards throughput or written storage.

The CLI allows configuring all writing arguments t8code exposes via the `t8_forest_write_netcdf_ext` function. These include the following:

- The IO mode, namely independent IO, collective IO, and the file-per-process mode described in section 3.1.
- The storage mode, namely chunked vs contiguous IO.
- The fill mode, which is used to debug NetCDF writing but is always overhead.
- The NetCDF version. These were listed in section 2.2. Note that still only CDF5 and netCDF4/HDF5 file can be supported.
- If file-per-process mode is used.
- The chunksize for the coordinates.

Additionally, the CLI allows specifying the number of element-wise data variables. Storage for the given number of variables is created and filled with pseudo random values. The created variables are given to the `t8_forest_write_netcdf_ext` function and exported

[34] UGRID Conventions (v1.0), *Accessed on 31 August 2022*: `https://ugrid-conventions.github.io/ugrid-conventions/`

as part of the NetCDF export. The number of element-wise data variables is also given to the selected scenario's forest creation routine in case it should influence the forest creation. The pseudo random scenario, for example, accounts for the element-wise variables in its size calculation to create a forest of roughly the given size independent of the number of element-wise variables.

The element-wise variable data is only created after the forest is partitioned. This way, it is known how many data points each process has to create for itself. Alternatively, you could create variable data before partitioning the forest and then use `t8_forest_partition_data` to distribute the element-wise variable data. The measurements taken by the CLI are unaffected by which data partitioning approach is used, as only the writing is measured. The approach where the data is generated where needed is chosen because it is strictly more efficient.

An example workflow using the CLI might look like the following depending on the use-case: Alice wants to know whether collective or independent IO is faster when for their use-case, which involves writing a 100 GB forest on a supercomputer with 2 nodes and a total of 96 CPU cores. They reserve a Slurm session with the following command:

```
salloc --nodes=2 --ntasks-per-node=48 --time=1:00:00 --constraint=scratch
```

Then they run the CLI with the following command:

```
srun ./t8cdfmark --num_element_wise_variables=10
↪  --pseudo_random:bytes=100000000000 --netcdf_version=netcdf4_hdf5
↪  --storage_mode=NC_CONTIGUOUS --mpi_access=NC_INDEPENDENT
```

The output is shown in listing 5 where **throughput** = **actual_information_bytes** ÷ **seconds** Alice repeats the benchmark, this time with `NC_COLLECTIVE` instead of `NC_-`

```
{"actual_information_bytes":79861115776,"seconds":2.546637235,
↪  "throughput_B/s":31359439294.46237}
```

Listing 5: Example results.json output of the CLI

`INDEPENDENT`. They see that the throughput is almost three times bigger and conclude that they should use this setting in their use-case.

The following is an overview of the steps the CLI takes internally:

1. Parse CLI arguments
2. Create forest according to selected scenario
3. Partition forest
4. Create element-wise data variables
5. Print storage size of forest, disregarding NetCDF metadata like attributes.
6. Save current timestamp
7. Export the forest as NetCDF
8. Calculate the passed time and throughput.

9. Print both as well and exit.

As for non-functional requirements, the CLI must have good performance since it will be run on supercomputers where any inefficiencies incur large costs.

5.2 Benchmark orchestration

The benchmark CLI described in section 3.2, as well as the t8cdfmark CLI described in section 5.1, that is used for all benchmarks presented here, executes one benchmark with specific settings. For comprehensive benchmarking, benchmarks with many different combinations of arguments need to be executed. To make this reproducible and scalable a small Python[35] script is created as part of t8cdfmark that ingests a configuration file, an example of which is shown in listing 3, and schedules the listed benchmark configurations by interfacing with Slurm. The benchmarks are prevented from interfering with each other by running them strictly sequentially. The following implementation options are available to achieve sequential behaviour:

- The Python subprocess module[36] is used to spawn `srun`[37] sub processes from the script. `srun`, as well as `subprocess.run` block until the job is complete, making the sequentialisation trivial. Making many small Slurm reservations like this however is discouraged by the GWDG's Slurm admins, because Slurm does not get all the scheduling information upfront. To mitigate this, salloc[38] can be used to run multiple `srun` invocations under the same Slurm reservation.

- The chosen implemented approach is to invoke `sbatch`[39] from the script. This way the benchmark runs are only scheduled in the script and it terminates almost immediately. To sequentialise the execution, each job, except the first, gets attached a dependency on the previous job. This approach is better, because Slurm gets the job information for all benchmark runs as soon as possible, making its scheduling tasks easier. Furthermore, individual jobs can be observed using the regular Slurm tools. Even though `sbatch` is used, we can avoid managing `sbatch` scripts ourselves by using the `-wrap` parameter and letting Slurm create virtual `sbatch` scripts as shown in listing 6.

- Another possible approach is to create a job array[40], which is suitable for executing many similar jobs. This however makes specifying benchmark parameters more complex and Slurm parameters would need to be specified separately.

For each benchmark run, the script creates a separate directory to contain the Slurm log file and disk usage information that is recorded as part of each run. The path contains all benchmark arguments. After each run, the created NetCDF files are immediately deleted, since they are not needed and to avoid accumulating disk usage across multiple runs. Additionally, only when the benchmark completed successfully, a file called

[35] python programming language, *Accessed on 10 November 2022*: `https://www.python.org/`

[36] python subprocess module, *Accessed on 10 November 2022*: `https://docs.python.org/3/library/subprocess.html`

[37] `srun` command reference, *Accessed on 10 November 2022*: `https://slurm.schedmd.com/srun.html`

[38] salloc, *Accessed on 10 November 2022*: `https://slurm.schedmd.com/salloc.html`

[39] sbatch, *Accessed on 10 November 2022*: `https://slurm.schedmd.com/sbatch.html`

[40] job arrays, *Accessed on 10 November 2022*: `https://docs.gwdg.de/doku.php?id=en:services:application_services:high_performance_computing:running_jobs_slurm:job_arrays`

```
sbatch --parsable --nodes=5 --ntasks-per-node=5 --wrap="module load
↪  netcdf-c openmpi; srun path/to/benchmark
↪  --num_element_wise_variables=10 --pseudo_random:bytes=100000000000"
```

Listing 6: Example `sbatch` usage similar to how the Python script invokes it.

"`success`" is created in the mentioned directory. Before scheduling a benchmark the script checks whether this file already exists and skips the benchmark if it does. This is useful for repeating failed benchmarks in a set of configurations.

5.3 Data collection

After a set of benchmarks completes, the job outputs, especially the measured times we are interested in, are still spread throughout the directory. A second script reproduces the directory tree created by the launch script when given the same configurations, and then walks this tree and extracts the measurements from the CLI's output files. t8cdfmark produces JavaScript Object Notation (JSON), which is parsed using Python's standard library. If the CLI contributed to t8code described in section 3.2 is used, the following regular expression may be used to achieve the same. Example scripts demonstrating this usage are part of the supplement in section 8.

```
The time elapsed to write the netCDF-4 File is: ([0-9]+\.[0-9]+)
```

While iterating the configurations the script outputs a CSV file with a header as shown in listing 7, that associates a set of benchmark arguments, including the repetition and the Slurm arguments, with the recorded time. The result is easy to parse for tools, for example by pandas[41], to allow data analysis and subsequently visualisation.

```
nodes,tasks_per_node,storage_mode,cmode,comm_mode,
↪  num_element_wise_variables,repetition,actual_information_bytes,
↪  seconds,throughput_B/s
```

Listing 7: CSV header of output from `extract.py`

6 Results

The results for strong scaling with respect to `tasks_per_node` and `nodes` are shown in fig. 2 and fig. 3 respectively. Notice that file-per-process generally performs better than non file-per-process. In both graphs there is no data for `NC_INDEPENDENT` with `NC_CHUNKED` because all corresponding experiments timed out after sixty minutes. Since they had at most 100 GB to write their throughput must have been less than 28 MB/s. This threshold is drawn in fig. 2 once for comparison. Preliminary experiments on smaller scale indicated similar results. For `NC_COLLECTIVE` with `NC_CHUNKED`, most experiments ran out of Random Access Memory (RAM), the cause of which remains to be investigated.

[41] pandas, *Accessed on 10 November 2022*: `https://pandas.pydata.org/`

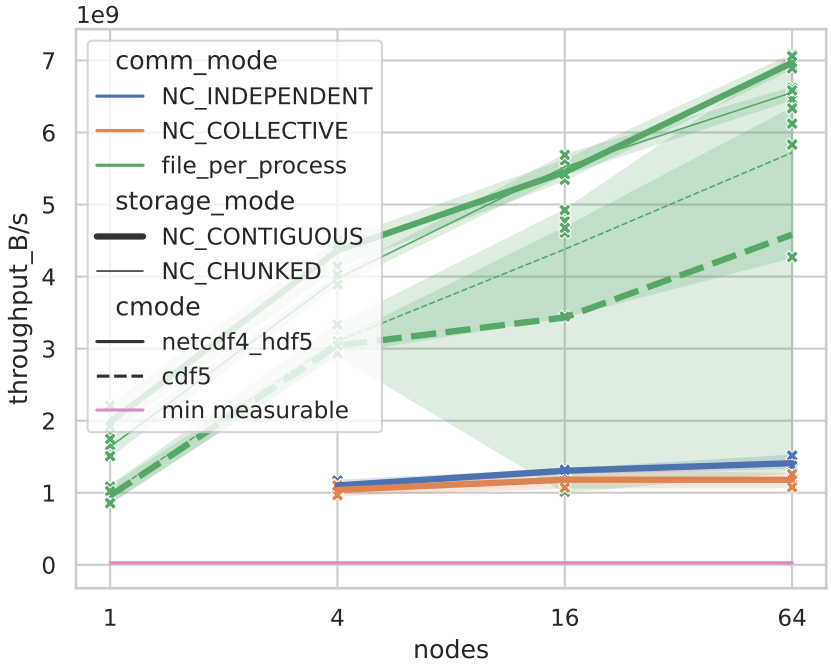

Figure 2: Throughput over node count with five tasks per node in a strong scaling scenario writing a forest roughly 85 GB in size for various settings. The lines show the mean over three repetitions and the error bands show the 95% confidence interval.

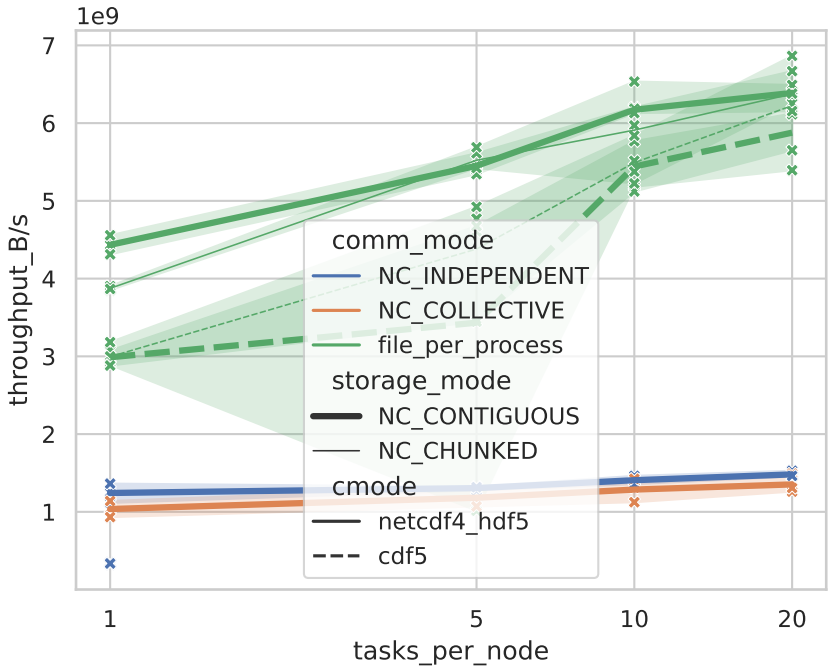

Figure 3: Throughput over tasks per node on 16 nodes in a strong scaling scenario writing a forest roughly 85 GB in size for various settings. The lines show the mean over three repetitions and the error bands show the 95% confidence interval.

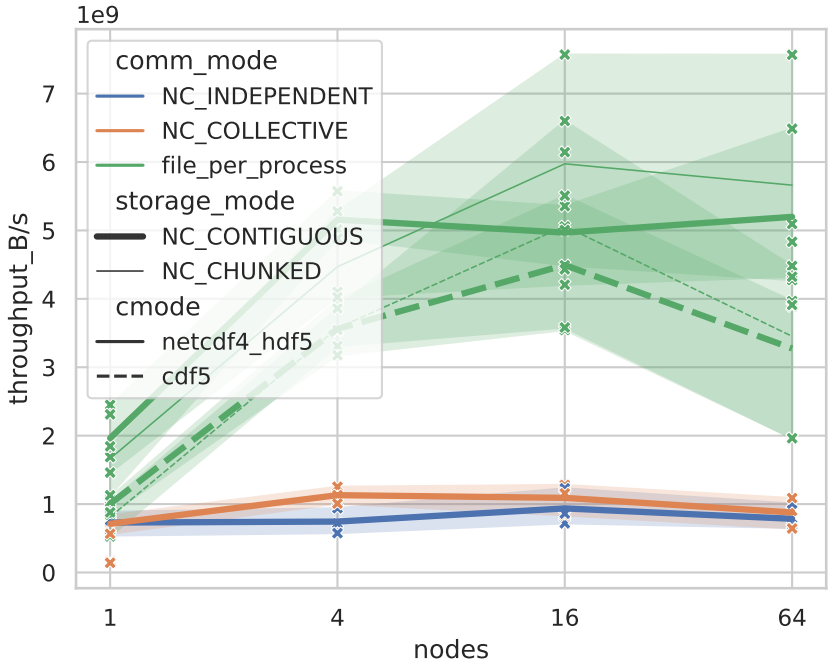

Figure 4: Throughput over node count with five tasks per node in a weak scaling scenario writing a forest between roughly 13 GB and 845 GB in size for various settings. The lines show the mean over three repetitions and the error bands show the 95% confidence interval.

The weak scaling results are shown in fig. 4. Notice that from 16 to 64 nodes, the file-per-process throughput notably decreases in most configurations. This may be due to the aforementioned filesystem contention that file-per-process IO is prone to suffer from for high process counts. A good writing configuration is expected to saturate at some maximum throughput to which it should stay close for high parallelism. t8code is expected to run with even more parallelism, where the throughput may quickly become worse for example with file-per-process mode and `CDF5` format, as indicated by the graph. From the results so far you can also conclude that `CDF5` is generally detrimental to file-per-process throughput compared to `netCDF4/HDF5`.

A possible explanation for why collective IO outperforms independent IO here, as opposed to in the previous strong scaling experiments, is that individual write calls are smaller. At the four node mark, here each process writes roughly 2.6 GB while in fig. 2 each process writes roughly 4.2 GB. Collective IO may have a performance advantage over independent IO when accumulating the smaller calls into larger writes.

6.1 Comparison with ESDM backend

The same sets of benchmarks were repeated with ESDM as a NetCDF backend. ESDM commit e49a9d9[42] and esdm-netcdf-c commit 0c6e722[43] was used for the ESDM bench-

[42]ESDM commit e49a9d9, *Accessed on 17 January 2023*: `https://github.com/ESiWACE/esdm/tree/e49a9d9ed08f25a6f31eac9b2c632a262282f91a`

[43]esdm-netcdf-c commit 0c6e722, *Accessed on 17 January 2023*: `https://github.com/ESiWACE/esdm-netcdf-c/tree/0c6e722221c09cefbfe5f0fd5984f4388de29a36`

marks. The results are shown in fig. 5, fig. 6, and fig. 7, where the focus is on comparing ESDM with the default NetCDF implementation. All experiments show the ESDM backend consistently outperforming the default NetCDF implementation. File-per-process comes out on top sometimes, but in fig. 7 it slows down at 64 nodes while ESDM throughput still rises. File-per-process is not expected to bring performance improvements in combination with ESDM since it implements a similar optimisation itself. Additionally, by using file-per-process, ESDM is deprived of context and information that may be useful to it. The impact can be observed in all experiments.

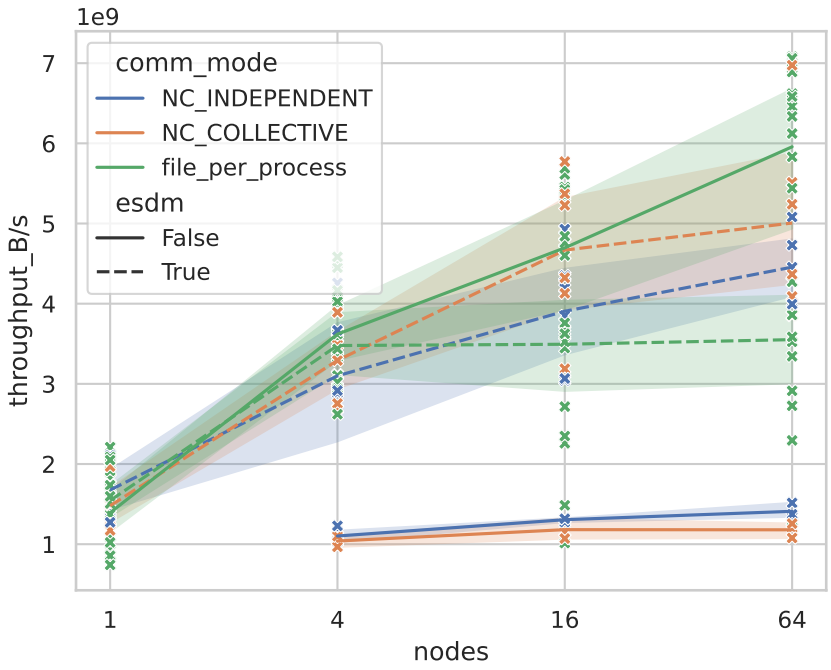

Figure 5: Throughput over node count with five tasks per node in a strong scaling scenario writing a forest roughly 85 GB in size. Only the communication mode and whether ESDM was used is differentiated. All other measurements are aggregated. The lines show the mean and the error bands show the 95% confidence interval.

6.2 ESDM settings comparison

Figure 8, fig. 9, and fig. 10 show the three sets of benchmarks as before, but this time, various settings in combination with the ESDM backend are compared. File-per-process is left out, because it was deemed uninteresting due to the reasons mentioned above.

7 Discussion

As for recommendations to t8code users, I would strongly advise against collective IO in combination with the chunked storage mode, since the performance is abysmal. Both collective and independent IO work similarly well with the contiguous storage mode. From these, I slightly prefer collective IO since it performs a bit better on large files in the realm

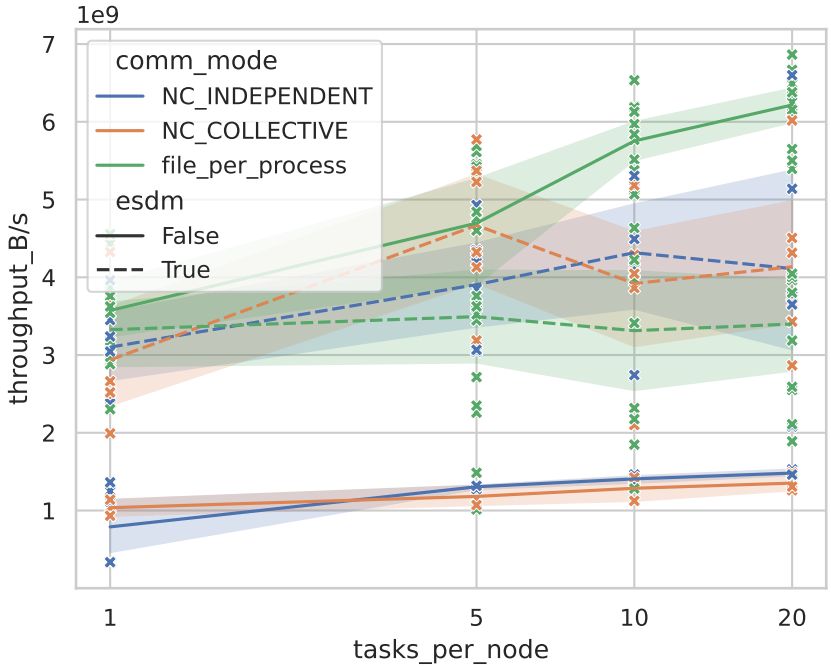

Figure 6: Throughput over tasks per node with on 16 nodes in a strong scaling scenario writing a forest roughly 85 GB in size. Only the communication mode and whether ESDM was used is differentiated. All other measurements are aggregated. The lines show the mean and the error bands show the 95% confidence interval.

of t8code use cases. That said, t8code's defaults, contiguous and independent[44], are perfectly fine. If possible, try using ESDM, since I measured up to five times speedup and consistently better throughput at little labor investment. It is hard to give a recommendation to ESDM users due to the inconsistent measurements but collective, contiguous IO appears to be a good default with no large shortcomings.

The implemented file-per-process performs well for the tested parallelism sizes but is probably unusable in its current form in practice, since it does not produce a single accessible NetCDF file like the other options.

7.1 Issues

There are few repetitions and a sparse grid coverage in the benchmarks. They are hard to produce due to their runtime and the amount of possible parameter combinations. Conducting these kinds of benchmarks require careful planning and cannot be iterated upon easily. In retrospect a higher tasks per node count in the weak scaling benchmarks would have been more realistic and therefore made for more representative benchmarks. This work also produced little data on chunked IO. Furthermore, the underlying hardware limitations were not sufficiently investigated in this work.

The fact that during the weak scaling benchmarks, the written storage is only a hint complicates the interpretation of the results. The throughput is accurately calculated, but the problem size is only approximately proportional to the parallelism.

[44] t8code NetCDF defaults, *Accessed on 29 January 2023*: `https://github.com/DLR-AMR/t8code/blob/fd2f70168672ccaf6abdb5a1a598d5a97aa03914/src/t8_forest/t8_forest_netcdf.cxx#L1246`

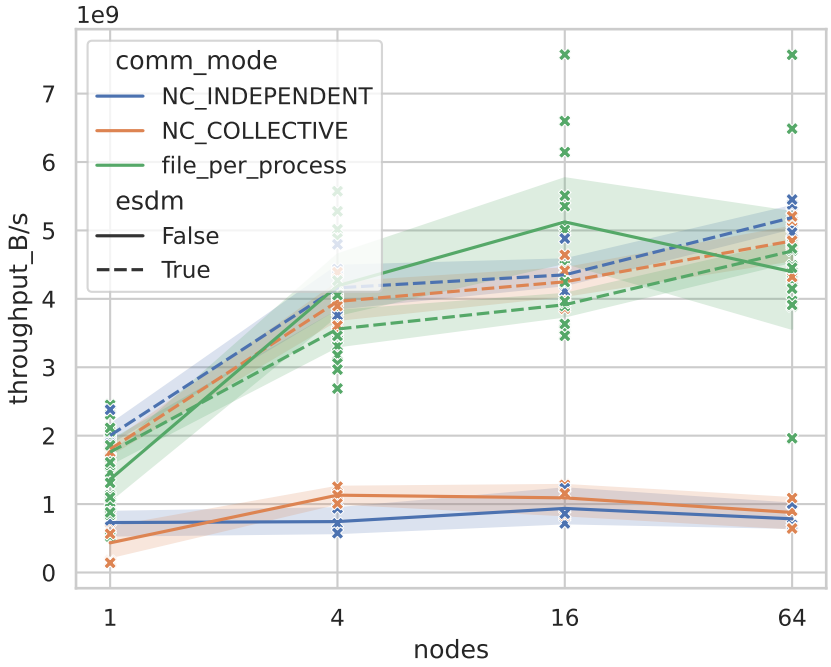

Figure 7: Throughput over node count with five tasks per node in a weak scaling scenario writing a forest between roughly 13 GB and 845 GB in size. Only the communication mode and whether ESDM was used is differentiated. All other measurements are aggregated. The lines show the mean and the error bands show the 95% confidence interval.

More realistic t8code scenarios as described in section 5.1 were not implemented. The benchmarks were solely conducted with the Open MPI - MPI-IO implementation. There are many MPI-IO implementations which may give different or better results. Not much consideration went into the ESDM configuration file. Its performance impact in this work is therefore not well understood.

8 Conclusion

In this work I successfully implemented file-per-process as an output mode in t8code, a contribution that was merged into a feature branch of t8code. A small configurable benchmark program and accompanying documentation was also accepted.

The custom benchmark suite t8cdfmark was realised as described in section 5, barring the refinement models apart from pseudo-random. Its usefulness was demonstrated by using it to produce the benchmarks in this work. To the best of my knowledge, a benchmark suit like t8cdfmark did not exist before this work. t8cdfmark is flexibly configurable and extensible to represent a variety of use-cases surrounding t8code. t8cdfmark and all created benchmark data is openly available.

The conducted benchmarks show the performance advantages of file-per-process IO as well as the ESDM backend. For example, ESDM shows a roughly a five times speedup out of the box over plain NetCDF when writing a 845 GB forest on 64 nodes. File-per-process output is shown to have similar performance advantages but does not produce the information in a useful format. Furthermore, I show that NetCDF's chunked IO should

be avoided and t8code has good default NetCDF settings.

In future work, t8cdfmark may be improved with an improved understanding of common t8code and other use-cases. More benchmark data with different foci may be collected, possibly on a different HPC cluster. If I understand correctly, the t8code team works on reducing the written information directly. The UGRID conventions do not exploit the inherent SFC structure t8code employs. Currently, a cube element for example occupies 268 Bytes only for its metadata. Most of this overhead is due to the node coordinates being stored explicitly, even though they could be derived from the tree structure. Such optimizations will be explored by the t8code team in future work and may bring large IO performance improvements.

Appendix

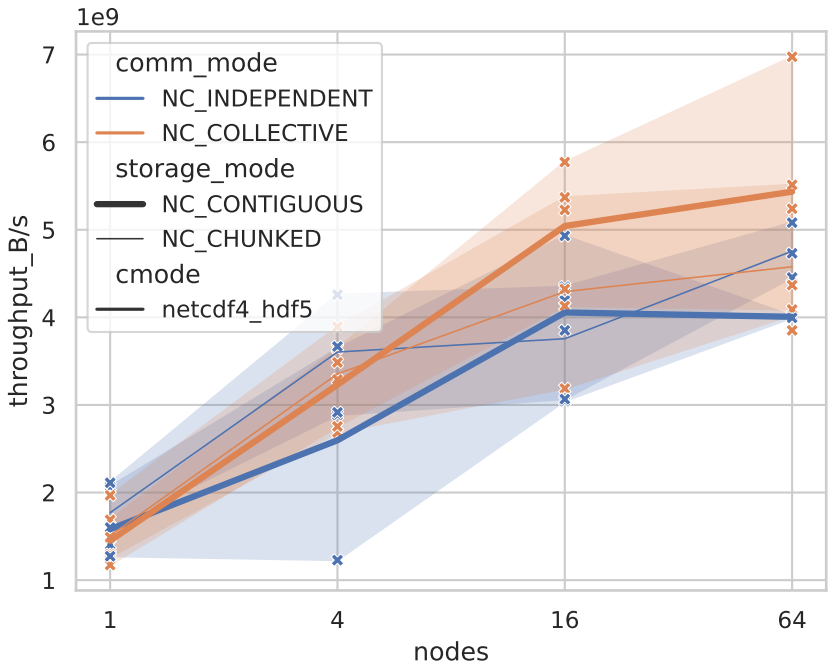

Figure 8: Throughput over node count with five tasks per node in a strong scaling scenario writing a forest roughly 85 GB in size for various settings using ESDM. The lines show the mean over three repetitions and the error bands show the 95% confidence interval.

The attached supplement includes all measurements presented here, the Jupyter notebook used to create the plots, the ESDM config file, and supplemental scripts that may be useful with the t8code benchmark CLI from section 3.2. For t8cdfmark, please visit `https://github.com/neoq/t8cdfmark`. To get the supplement, save the following link's target as a `.tar.gz` file: `supplement.tar.gz`

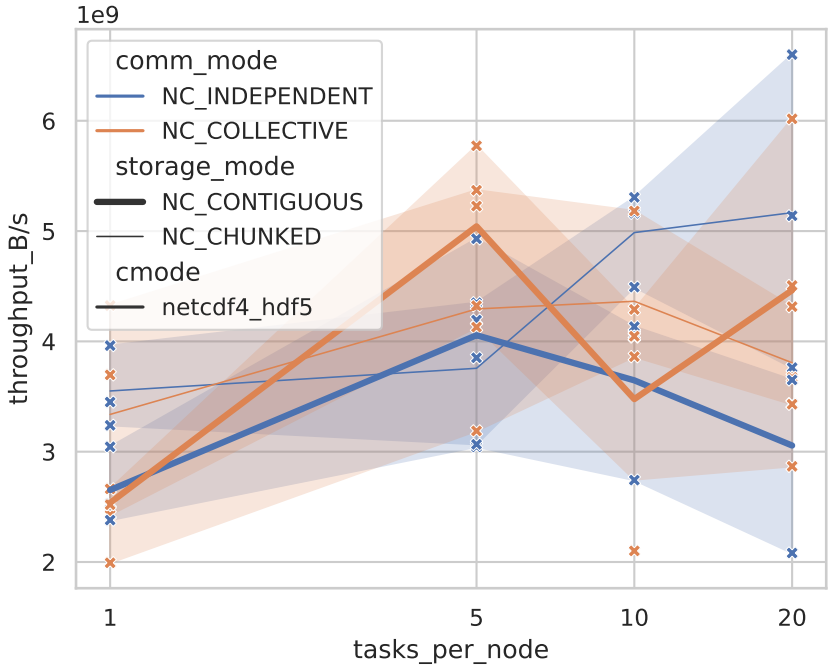

Figure 9: Throughput over tasks per node on 16 nodes in a strong scaling scenario writing a forest roughly 85 GB in size for various settings using ESDM. The lines show the mean over three repetitions and the error bands show the 95% confidence interval.

References

[22] *NetCDF*. Nov. 2022. DOI: https://doi.org/10.5065/D6H70CW6.

[Beh+19] Babak Behzad et al. "Optimizing I/O Performance of HPC Applications with Autotuning". In: *ACM Trans. Parallel Comput.* 5.4 (Mar. 2019). ISSN: 2329-4949. DOI: 10.1145/3309205. URL: https://doi.org/10.1145/3309205.

[Hol+22] Johannes Holke et al. *t8code*. Version 0.10.0. July 2022. URL: https://github.com/holke/t8code.

[Joh18] Johannes Holke. "Scalable Algorithms for Parallel Tree-based Adaptive Mesh Refinement with General Element Types". PhD thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, Dec. 2018. URL: https://hdl.handle.net/20.500.11811/7661.

[KP20] Julian M. Kunkel and Luciana R. Pedro. "Potential of I/O Aware Workflows in Climate and Weather". In: *Supercomputing Frontiers and Innovations* 7.2 (July 2020). DOI: 10.14529/jsfi200203. URL: https://superfri.org/index.php/superfri/article/view/309.

[Li+03] Jianwei Li et al. "Parallel NetCDF: A High-Performance Scientific I/O Interface". In: *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. SC '03. Phoenix, AZ, USA: Association for Computing Machinery, 2003, p. 39. ISBN: 1581136951. DOI: 10.1145/1048935.1050189. URL: https://doi.org/10.1145/1048935.1050189.

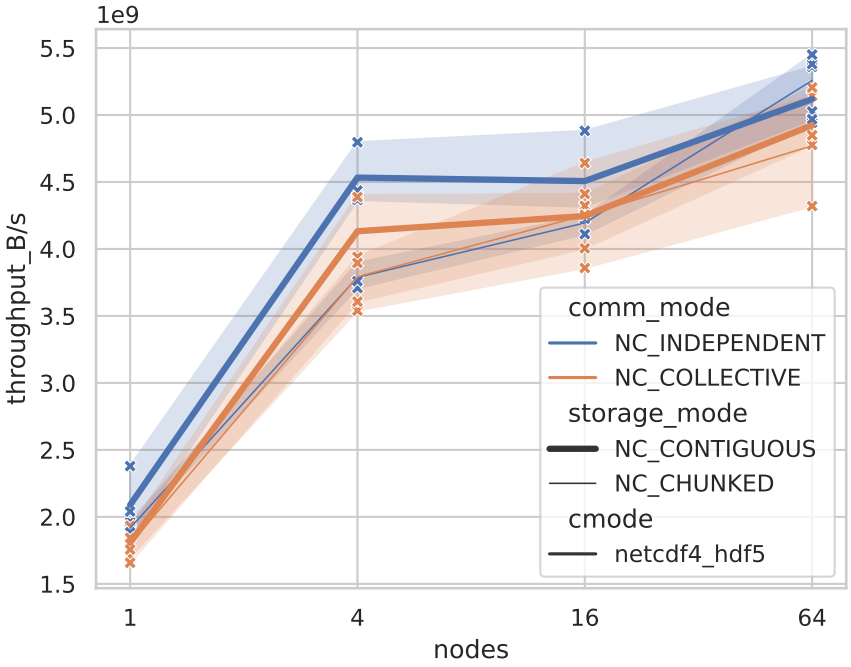

Figure 10: Throughput over node count with five tasks per node in a weak scaling scenario writing a forest between roughly 13 GB and 845 GB in size for various settings using ESDM. The lines show the mean over three repetitions and the error bands show the 95% confidence interval.

[Mes09] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 2.2*. Sept. 2009. URL: `https://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`.