The 8th European Congress on Computational Methods in Applied Sciences and Engineering
ECCOMAS Congress 2022
5–9 June 2022, Oslo, Norway

ON HIGHLY SCALABLE 2-LEVEL-PARALLEL UNSTRUCTURED CFD

Jens Jägersküpper and Daniel Vollmer

German Aerospace Center (DLR)
Institute of Aerodynamics and Flow Technology
Center for Computer Applications in AeroSpace Science and Engineering
38108 Braunschweig, Germany
Jens.Jaegerskuepper@DLR.de

Key words: Computational Fluid Dynamics (CFD), High-Performance Computing (HPC), unstructured meshes, parallel computing, parallel programming, exascale

Abstract. Exascale HPC systems are just about to become available. Such enormous simulation capabilities from the hardware perspective, however, require software that can effectively make efficient use of this massively parallel hardware. For the domain-decomposition based algorithms generally used for CFD, it has become apparent that running one MPI process, i.e. one domain, per CPU core is no longer apt when there are hundreds of cores available per cluster node. There are just too many processes per node that need to communicate with other remote processes. Message aggregation is thus a key aspect of highly scalable parallel codes. A 2-level parallelization featuring a shared-memory level in addition to the MPI process level seems a promising solution. The CODA (CFD for ONERA, DLR, Airbus) software for high-fidelity compressible-flow simulations of industrial configurations implements such a 2-level domain-decomposition hybrid-parallel approach. The processing of unstructured meshes is particularly challenging with respect to load balancing and non-linear data access. For maximum parallel scalability, CODA's parallelization not only features overlapping communication with computation on the process level, but also a dedicated Single Program (on) Multiple Data (SPMD) programming model for the shared-memory level. Here, the design principles of this parallelization concept are outlined, followed by details concerning its implementation in the *Flucs* infrastructure (FLIS), which has become part of CODA. Moreover, results of scalability studies with CODA are presented, which impressively demonstrate the extreme scalability realized with this parallelization approach. The studies were performed on two distinct HPC clusters, one based on Intel's Xeon Scalable Processor, the other one based on AMD's EPYC.

1 Introduction

Finally, exascale systems are becoming evailable these days. With CPUs featuring 64 physical cores, for instance, hundreds of hardware threads need to be utilzed per cluster node. “Hybrid” programming models that reflect not only the inter-node level parallelism, but also the intra-node parallelism have thus come up. These models are sometimes coined “MPI+X”, where “X” denotes some shared-memory model, often utilizing multi-threading. The advent of a “shared memory model” in version 3 of the message passing interface standard (MPI-3) may be taken as an unmistakable indicator that even the advocates of “MPI is all you need for parallel programming” have come to the conclusion that message passing is not sufficient to address such HPC architectures. Time will tell, however, whether choosing ”MPI” for ”X” in ”MPI+X” (cf., e.g., [6]) will win the day. Currently, OpenMP seems a more favorable – or, at least, a more popular – choice. In fact, alternatives to MPI for the abstraction of distributed memory are also available. In particular the approach of a “partitioned global address space” (PGAS) has gained some interest, one API being “GASPI”, cf. [7], another one “OpenShmem”.

For domain-decomposition based algorithms, it has become apparent that running one MPI process, i.e. one domain, per core (or even hardware thread) is no longer apt when there are hundreds of cores available per cluster node. Even though MPI implementations use zero-copy message passing within a cluster node, there are too many processes per node that need to communicate with other remote processes (some of which may actually be located on the *same* remote cluster node), resulting in excessive use of network resources. In particular, message rates of network hardware have not kept (and will not keep) up with the rate of increasing core count per node. So, message aggregation has become important to lower the pressure on the network. The challenge with this is to have message aggregation scale with the number of domains/threads/processes involved. If message aggregation is processed serially, such an approach is unlikely to scale well (on the node level) – a simple consequence of Amdahl’s law.

The application domain considered here is mesh-based computational fluid dynamics (CFD). With increasing compute power, the numerical simulation of flows has become a major design tool, in particular in the automotive and aerospace industry. Scale-resolving time-accurate simulations of non-stationary flows are still a computational challenge, however, in particular for high-Reynolds-number flows. Transforming the raw compute power of HPC hardware into application/simulation performance has actually become harder: Parallel scalability of most applications is rather limited today such that only a moderate amount of compute resources can be reasonably assigned to a single CFD simulation. Thus, lifting CFD software (most importantly, its parallelization) to deliver peta- and upcoming exa-scale performance is an issue actively addressed – in both, academic and industrial research.

The parallelization of CFD algorithms usually relies on some kind of domain decomposition, i.e. splitting the computational domain in multiple parts, called “domains”. In order to couple the individual computations on these domains to each other, copies of field data that belong to one domain exist at other domains, which are called “ghost” or “halo” data. For mesh-based CFD, this data consists of flow states in cells (e.g. finite volumes) of the mesh or derived data like gradients, for instance. Ghost-cell exchanges arise whenever some form of domain decomposition is used in combination with short-range interaction, e.g. a stencil operation to compute gradients or the computation of numerical fluxes over faces between any two cells to evaluate the spatial discretization (for a given field to obtain the corresponding residual). For best possible parallel efficiency/scalability of such distributed operations, the update of halo data should affect the parallel execution as little as possible, of course. This is the aim of the work presented herein.

2 Motivation and Related Work

At the German Aerospace Center (DLR), CFD codes have been developed for decades. Several of these codes are in production, i.e. in regular industrial use. One such code is the DLR

TAU code which is mainly used to simulate external aerodynamic flows using a 2nd order finite-volumes discretization, parallelized “MPI-only” using classical domain decomposition with a single layer of ghost cells. In addition to functional enhancements w.r.t. simulation capabilities over the years (cf. [11]), TAU received several optimization activities w.r.t. its HPC ability. In particular, in the national projects “HICFD” [4] and “GASPI” [1], a TAU prototype was created that featured shared-memory parallel processing of each individual domain (within a process) via a task-based threading model, resulting in a 2-level parallelization [9]. This parallelization showed very promising results for the common HPC cluster architecture of that time, namely hexa/octo-core Intel Xeon processors connected with an InfiniBand network. Performance evaluation for the Intel Knights Corner (KNC) architecture, however, uncovered bottlenecks due to the task dispatching/synchronization, resulting in poor scalability for KNC. Even though locked memory accesses (to organize the threads) were used only at low frequency, the resulting coherence traffic over the ring bus connecting the KNC cores’ L2 caches was identified as the root cause for this shortcoming. Eventually, these findings and the enormous effort of such a refactoring, resulted in the decision to not adopt this parallelization model into the production version of TAU. Related considerations concerning parallelization approaches to domain decomposition and/or hybrid strategies can be found for instance in [13, 3, 5, 2].

In 2012 DLR decided to kick off the development of a new CFD code, the designated successor of TAU. This gave the extraordinary opportunity to design a parallelization, actually a comprehensive HPC concept, from scratch. HPC was only one design driver, though. Further drivers were: strong fully implicit schemes for improved algorithmic efficiency, higher-order spatial discretization (Discontinuous Galerkin method enabling hp-adaptation) in addition to finite volumes with maximum code sharing, improved integration into Python-based multi-disciplinary process chains, modularity, cf. [10, 12]. The actual coding started late 2013, and the full parallelization has become available at the turn of the year 2014/15 as part of the infrastructure of DLR’s new “Flexible unstructured CFD software” *(Flucs)*. In 2017, DLR has provided *Flucs* (including its *Flucs infrastructure (FLIS)* with the newly designed and implemented 2-level parallelization) as initial contribution to the cooperation between Airbus, ONERA, and DLR for a common CFD capability and has continued development in CODA.

Here the focus is on this 2-level domain decomposition which addresses distributed memory and shared memory differently, but in a highly coordinated way. Actually, the shared memory level is the interesting one, and it will thus be discussed in detail in the following Section 3. In Section 4, details are given how this design is actually implemented in the (source) code. Results on the parallel scalability of this implementation as currently available in CODA are then presented in Section 5. Section 6 ends this work with some concluding remarks.

3 Design

The credo of the parallelization presented here is simple: synchronize as little as possible. This means avoiding global barriers, performing synchronizations as locally as possible (namely as local as the dependencies implying the need for synchronization), and trying to prevent accumulation of waiting times. These goals are valid for both the distributed-memory parallelization as well as the shared-memory one – the driving aim being maximum scalability of the code, i.e., being able to use as many processing elements for a simulation as possible. The idea behind this aim is to enable users of CODA to speed-up their simulations’ time-to-solution virtually at will.

Classical domain decomposition for distributed memory uses halo cells to have remote data locally available (for stencil/face operations). Consequently, the total memory footprint of a decomposed computational mesh increases with the number of domains (MPI processes). With the increasing total memory footprint the total number of memory accesses during a distributed stencil operation over the distributed mesh also increases. This is reason enough for a limited parallel scalability and one major reason to have a different level of parallelization address the shared-memory level of the hardware: Namely an approach that does *not* use halo copies.

Note that using an MPI-only implementation of domain decomposition to utilize all the cores of a single chip actually constitutes a shared-memory parallelization: an MPI implementation is likely to offer a transport layer that makes use of shared-memory segments to "pass" messages zero-copy, without actually involving the network layer. As a consequence, one would expect a shared-memory implementation to show at least a parallel efficiency as good as the "misapplied" MPI-based one. If this basic test fails, i.e., if the shared-memory parallelization performs noticeably worse than an MPI-only one, this is a clear indicator that the shared-memory implementation offers room for improvement. For a 2-level domain-decomposition, using the coarse-level decomposition with each process running only a single thread, represents a benchmark for the efficiency of the fine-level one running only a single process but fully threaded. For a homogeneous HPC architecture, running one process per chip (socket) should result in maximum parallel efficiency – as cross-socket cache-coherence traffic between threads of the same process running on different sockets usually affects the scalability significantly. For chips that inherently feature non-uniform memory access (NUMA), however, the effect of keeping cache data coherent between multiple uniform memory access (UMA) regions may be crucial, potentially requiring running multiple processes per socket. Having said that, the numerous publications reporting that retrofitting a shared-memory parallelization to an existing MPI-only parallelized (legacy) code did not result in an improved parallel scalability (e.g. [8, Sec. 4.8]) show how important it is to do things right – here the combination of distributed- and shared-memory parallelization. In the present work, both levels were co-designed in a holistic concept from scratch.

3.1 Distributed-memory level (coarse, processes)

Though the focus will be on the shared memory level, some details on the distributed memory level (which set it apart from classical implementations) are presented first. As ghost-cell based domain decomposition implies local dependencies, each between two domains (processes), classic implementations use point-to-point communication to update the so-called "halo" of a domain consisting of the ghost cells (copies of remote data), commonly using `MPI_Isend` and `MPI_Irecv`, finalized by an `MPI_Waitall`. Accumulation of waiting times, however, should be avoided. Note that halo-data access only happens during loops over the faces of a mesh, whereas loops over the cells of a mesh only touch domain-local data. To *overlap the communication of halo data with computation*, in order to minimize waiting times on the distributed-memory level, halo data to send to other domains (processes) is gathered and sent off using non-blocking semantics at the beginning of a face loop. To minimize waiting times also on shared-memory level, all threads contribute to this part of the halo exchange (to prevent a serial operation that would limit the scalability). For a face loop, inner faces (each connecting two domain-local cells owned by the same thread) are then processed first (by that owning thread). While processing these inner faces, halo data is being transferred. Before halo-touching faces (each connecting a local cell with a ghost cell) are finally processed, the arrival of halo data needs to be ensured. The processing of receiving and scattering halo data is also multi-threaded to prevent a serial bottle neck that would limit scalability. Details on how this is done follow in the next section.

3.2 Shared-memory level (fine, threads)

In fact, the multi-threaded halo-exchange logic outlined above is a key feature to the overall scalability of the 2-level domain decomposition. It was one design driver of the shared-memory parallelization of the processing of a domain by multiple threads. A further design driver was data locality. *For maximum data locality, each cell is "owned" by a unique software thread which is exclusively pinned to a hardware thread when running the code.* This mapping of cells to threads is achieved by a static decomposition of each domain (mapping one-to-one to a process) into as many sub-domains as threads are to be used, i.e., a *one-to-one mapping of threads to sub-domains* is employed likewise. This allows the use of the Single Program (on) Multiple Data (SPMD) model known from MPI programming also for the threading level. Moreover, writes to a cell's data can now be arranged such that only very coarse synchronization is needed, because these writes are performed by the owning thread only – with the one exception being the receiving of halo data, cf. below. All other threads may access this cell's data solely read-only. This means that invalidation of a cell's data in the cache-memory hierarchy only takes place in a fixed path through the memory hierarchy: starting at the L1-cache the respective thread is attached to, down to main memory. For (memory) architectures that do read-for-ownership also for read-only access, this may not seem important. For architectures that distinguish writes from reads, however, this can have a noticeable effect. For HPC systems, this design feature may well be beneficial, depending on the actual cache-coherence protocol of the respective hardware. Furthermore, data migration between core/thread local caches (usually L1 and L2) is minimized, addressing the apparent change from compute-centric to data-centric architectures – namely the fact that computation is becoming relatively cheap, whereas data movement is becoming relatively costly (particularly w.r.t. energy considerations).

For the static sub-domain decomposition (of each domain), load balance is (again) essential. Not only for the scalability of the shared-memory level itself, but also to prevent the propagation and accumulation of waiting times to/on the coarse/process level. Though comparatively little work is done in loops over the cells, the sub-domains (into which each domain is partitioned) should nonetheless also be equally sized w.r.t. the number of cells. The mapping of cells to sub-domains partially implies a mapping of faces to threads, namely a partitioning of the faces of a sub-domain: Each face for which both touching cells belong to the same sub-domain is obviously assigned to the thread to which the sub-domain is mapped one-to-one. Of course, the number of faces each thread is processing in a face loop should also be as balanced as possible, in particular since most computation in a face-based unstructured-mesh CFD code happens in face loops. A face between a domain-local cell and a ghost cell is to be processed by the thread to which the cell is mapped. Though finding a perfect partitioning of the cells satisfying both balancing needs is an NP-hard problem (w.r.t. mesh size), established heuristics exist.

The interesting faces, however, are those between any two cells that respectively belong to different sub-domains. Here data races could occur if a thread owning one cell were to also update data attached to the other cell (which is owned by a different thread). If one follows the above "local write-access only" design principle, there is only one option: such faces need to be processed twice, namely by both threads (respectively owning one of the two cells). Note that this actually results in a 3rd objective of the sub-partitioning: minimizing the number of such faces. A further motivation for this approach is that it significantly relaxes synchronization needs among the threads, which would otherwise be necessary to prevent data races. Recall that minimizing synchronization has been one of the design goals to maximize parallel scalability.

This might read odd at first, as it seems to foil the apparent idea of a hybrid parallelization. As cross-sub-domain faces are computed by both threads, the computational effort actually equals the one of a single-level decomposition in as many domains as there are now sub-domains. This is compute-centric thinking, though. With HPC going data centric, compute power is expected to abound. The additional shared-memory level described above does *not* contribute to the memory foot-print since ghost cells are only used on the coarse (process) level. Here redundant computations are deliberately traded for data locality and thread-synchronization savings.

An important aspect that has already been touched on above is *message aggregation for the halo-data exchange.* The halo communication pattern is determined by the coarse-level decomposition. When it comes to the packing of cell data to send to neighboring processes (domains), each thread copies those data attached to owned cells into a send buffer, targeting some neighboring process p that needs copies of the cells' data in its halo. Offset tables for this multi-threaded packing are precomputed. The one thread that finishes its contribution to the send buffer for p last, immediately initiates the data transfer to p (non-blocking). Notice that there is *no barrier.* Each thread can immediately start the processing of inner (sub-domain local) faces. There are actually *no waiting times* at all in these steps: gather halo data in communication buffers, initiate halo-data transfer, and process inner faces. Before halo-touching faces can be processed, however, halo-data exchange must be finalized. Deviating from the SPMD paradigm, the multi-threading of the receiving is done in a task-based manner: If there are n neighboring domains (processes) to receive halo-data from, there are n tasks: checking/making sure that halo data has physically arrived from a particular neighboring process and unpacking the halo data from the respective communication buffer into the local cell-data structure.

Aside: Actually, a performance tweak is possible here: Gathering of the halo-data at sender side can be organized such that no scattering is necessary at receiver side. So, a contiguous `memcpy` suffices to move halo data from the communication buffer. In fact, in-situ receiving is possible, avoiding any copy of incoming halo data. This is currently not implemented in FLIS due to an abstraction layer enabling the use of different network communication libraries.

So, after having sent off halo data and subsequently having processed the owned inner (sub-domain local) faces, each thread immediately starts looking greedily for pending receives to be finalized. When the receives have been finalized, the processing of a thread's halo-touching faces can begin – once halo data (for that thread, i.e. sub-domain) is marked ready by the halo-data receiving threads. (Note: Due to the greedy tasking, the order of these receives is arbitrary.)

In contrast to the sending, the multi-threaded receiving results in the need for thread synchronization and, as a consequence, potential waiting times. The simple solution would be a thread barrier right before the threads start with processing of halo-touching faces. The actual dependencies are local, though. Namely, each thread merely depends on the arrival of halo data from those processes that send halo data for those ghost cells that this thread actually touches.

As these dependencies are fixed (due the static process/domain neighborhood and the static subdomain decomposition), a minimum amount of thread synchronization can be established using "local" as well as "relaxed" synchronization primitives. How these are actually implemented is detailed in the following Section 4.

Though not the focus here, sequential/serial performance is optimized too, and one important aspect of this is data layout. Faces and cells are sorted such that indirect access to cell data during a face loop – for each face, data of the two connected cells is accessed – is blocked w.r.t. to space and time. Temporal blocking means that all the faces that touch a particular cell (namely the face-indices) are as close as possible in the sequence the faces are processed. Spatial blocking means that the data attached to cells that are subsequently accessed during a face loop are as close as possible in memory, which is obtained by arranging the cell-indices accordingly. The aim of this "grid renumbering" is an optimized cache utilization, cf. [8, sections 4.1 and 5.1.3]. This relates to "bandwidth optimization" of a matrix in computational (sparse) linear algebra.

4 Implementation

In contrast to TAU, which is essentially a C89 code, FLIS has originally been coded in C++11. The reason for choosing C++ is templates. These are used due to performance considerations and, in particular, to maximize source-code share for finite-volumes discretization and Discontinuous Galerkin method in CODA. A main reason for choosing the 2011 version of C++ at that time was multi-threading. It was the first version to support multi-threading in the core language. It allows the expression of formally correct multi-threaded C++ code, in particular via the possibility to express ordering dependencies among threads and their accesses to (shared) memory – a very important aspect when coding custom-made thread-synchronization primitives. This is particularly important when the code is supposed to be portable w.r.t. different architectures that each may use different hardware memory models. These days, C++17 is used for CODA (and FLIS).

Making use of C++ functors to specify element/face local operations, a kind of domain-specific language (DSL) is provided. This allows for expressing cell- and face-operations, which are then processed (in some unspecified order) in parallel over/for the complete mesh. A functor's `operator()` defines what is to be done for each cell or each face. Together with C++ "lambda expressions", this allows quite concise source code. In particular, there is only a single place in the source code where the actual mesh-looping logic is implemented. Someone coding a new CFD functionality via this DSL, however, will never get in touch with that part of the source code (which is rather elaborate). There, all the required synchronization is implemented – for the coarse level concerning the halo-data exchange as well as for the fine level concerning the multi-threading described in detail in Section 3. Moreover, additional logic is implemented in the "mesh looper" which spots whether halo donor data has changed, necessitating a halo exchange, or not. This is realized via accessor methods in the cell-data container "class template" which distinguish `const` access from write/update access. This of course requires consistent read and read-write accesses on all processes. (And to avoid thread-synchronization, these modification flags are in fact thread-local and therefore require consistent access also on the sub-domain level. This is achieved by declaring the intent by obtaining either a constant or a mutable pointer to the data before executing the actual loop.) By this, superfluous halo exchanges are avoided which might otherwise occur when a sequence of several face loops – potentially interleaved with a bunch of cell loops – is executed without updating halo donor-data, but merely computing temporary/intermediate field values.

Coming back to the 2-level domain decomposition, note that the coarse level decomposition is actually adopted from the surrounding MPI-parallel, Python-based, multi-discipline analysis and optimization (MDA/O) framework FlowSimulator ("FS") [14], namely its data manager "FSDM". FS(DM) features interfaces to the well known graph-based partitioning tools ParMetis and Sandia's Zoltan, which offer good load balance also and particularly for *un*structured meshes. For a cell-centered code like CODA, each cell maps 1-to-1 to a vertex of the graph to be partitioned, each face of the mesh to an edge of the graph. The second-level decomposition into sub-domains is done by domain-local calls to the Zoltan library, i.e., each process runs a separate (purely local) Zoltan instance. As an alternative, Metis can be used for sub-decomposition. Recall the three objectives of this sub-decomposition from Section 3: load balance w.r.t. number of cells as well as faces, and minimization of the number of faces that touch two different sub-domains. By choosing the number of faces of a cell as the weight of the graph-node corresponding to that cell (and unit weight for each mesh-face, i.e. graph-edge), load-balanced face-loops are favored over cell-loops. With minimizing the number of cut-edges, the amount of redundant (double) face calculations is minimized, cf. Section 3.

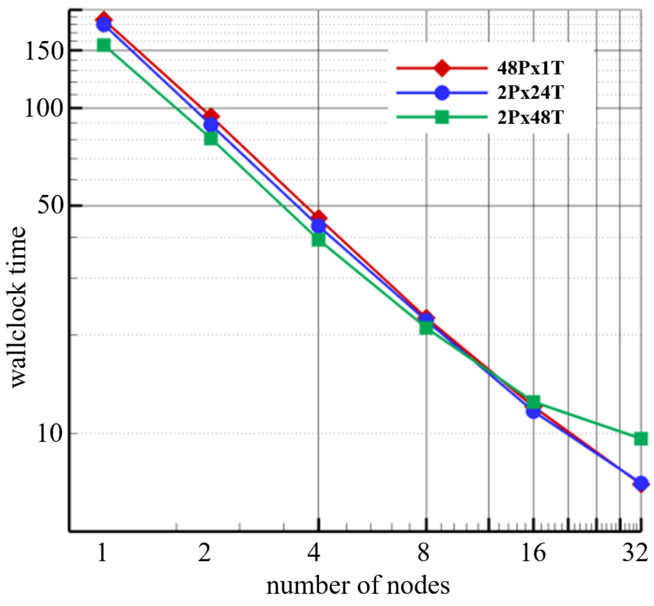
Once the 2-level decomposition has been established, when the control flow traverses from the MPI-parallel single-threaded SPMD-style FS/Python script into the SWIGed CODA code, threads are immediately forked. The threads remain forked until the control flow is about to return to the Python level – recall that the SPMD paradigm is applied with thread-synchronization reduced to a minimum. In fact, a single(!) OpenMP `parallel` region is used. (As no OpenMP work-sharing constructs are used, any other technology, e.g. "pthreads", would also do.) A desirable property of most OpenMP implementations is, however, that the non-master threads are not destroyed, but merely paused during the course of serial (i.e. master-thread only) parts in-between `parallel` regions. So, depending on the overhead of switching between parallel and serial regions, switching between single-threaded FS/Python level and multi-threaded CODA level should not be too frequent. As a consequence, CODA's time integration is coded in C++, not in Python. (One is free to implement a custom physical-time stepping in Python, of course.)

In the SPMD threading approach, different threads may be in different functions/methods at the same time (as there is no implied/implicit thread synchronization), for instance due to some load imbalance or external effects. One of the situations where some kind of synchronization is necessary, is when shared resources need to be allocated and/or initialized. To minimize waiting times, this should be done by the thread that arrives first, obviously. Also obviously, all other threads must wait until the first thread has finished. OpenMP's `single` work-sharing construct is sufficient for such a situation – but it imposes much more synchronization than necessary: At the implied barrier, waiting times accumulate unnecessarily. For instance, assume that the first thread is way ahead, such that it has finished initializing the shared resource already when the second thread arrives. Then none of the threads should do any waiting at all, meaning zero synchronization overhead! Such a "relaxed" synchronization is available in FLIS: The function `IsFirstOrWait()` immediately returns `true` to the first thread. Once this one thread calls `UnblockWaitingThreads()`, the function unblocks and returns `false` to all other threads. (Note that if this construct was to be used only once, it would be semantically equivalent to calling `std::call_once` using a `std::once_flag`.) The implementation for CODA supports multiple calls to `IsFirstOrWait()` on the threads' control flow which may indeed overlap, i.e., at a time, different threads may be in different calls. Each thread has a local "epoch" counter which is compared to a shared, "global" counter. The shared counter is a `std::atomic` which is accessed specifying appropriate `std::memory_ordering` semantics. So, the `IsFirstOrWait()` described above realizes a producer-consumer model. In the situation just considered, there is one producer and several consumers. For sending off halo data, however, the pattern is inverted: multiple threads "produce" data into the send buffer (for some neighboring process), where the one thread that finishes its contribution last "consumes" this data by initiating the transfer. This synchronization pattern is implemented analogously.

Another implemented thread-synchronization primitive is a non-blocking barrier, comparable to `MPI_Ibarrier` on the process level. Such a barrier is somewhat "relaxed" as the non-blocking arrival at the barrier is separate from the final blocking wait. Note that a thread blocks in the `wait()` merely until all other threads involved have indicated arrival. This may be much earlier in the program flow, depending on the amount of "independent" work that a thread encounters in-between arrival and final wait. This primitive is used to prevent the threads from deviating too much (recall SPMD), to protect against data races. In CODA, arrive() and wait() are quite distant for such non-blocking barriers. So, usually *no* waiting times accumulate at such non-blocking barriers in CODA (unless a software thread is preempted by the OS to an unusual extend). *Aside:* C++20 includes `std::latch` and `std::barrier` which do provide separate methods for arriving vs. waiting.

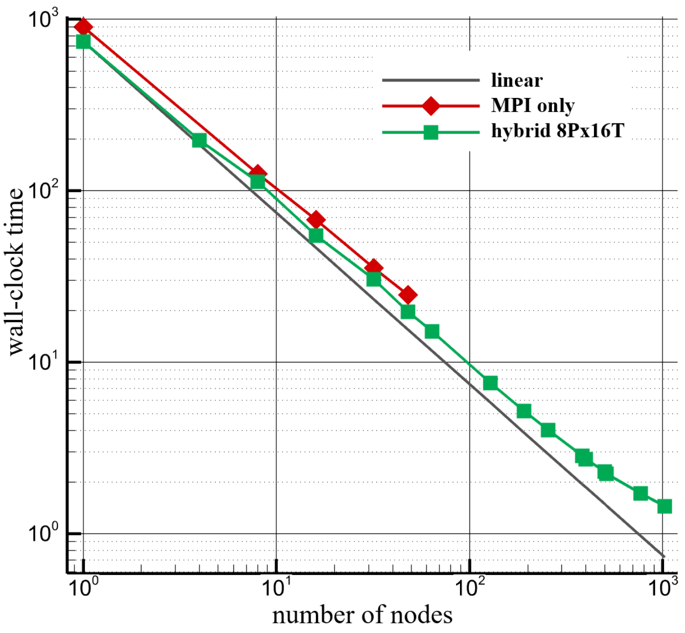
5 Scalability Results

During the design and implementation phases of the shared-memory parallelization described above, in addition to the common/dominating Intel Xeon architecture of that time, also IBM's BlueGeneQ architecture as well as Intel's Xeon Phi were considered. Very good parallel scalability was observed for these architectures – which have been discontinued by today. So, we concentrate on recent architectures here: Intel's Xeon Scalable Processor (SP) and AMD's EPYC processors. Both HPC clusters considered here feature dual-socket nodes and Mellanox-based InfiniBand network. The first one is the "JUWELS cluster module" located at the Forschungszentrum Jülich (FZJ). This cluster comprises over 2,500 nodes, each with two Intel 24-core Xeon Platinum 8168 CPUs. Due to the last-level cache (LLC) design of this architecture, which employs a common L3 cache for all cores of a chip (i.e. single UMA domain) the 2-level domain decomposition is run with one MPI process per socket, each running 24 threads (when not utilizing Hyperthreading) or 48 threads, each exclusively bound to one physical or logical core, respectively. To focus on the strong parallel scalability of the 2-level parallelization presented here, a tiny unstructured mesh with only about 1 million elements (hexahedrons and prisms) is considered. Namely, a classical ONERA M6 wing case is run: steady transonic flow using the Reynolds-averaged Navier-Stokes equations (RANS) with the 1-equation turbulence model by Spalart and Allmaras ("negative" formulation) as physical model. A 2nd-order finite-volumes scheme is applied as spatial discretization, time integration is done by a 3-stage explicit Runge-Kutta ("eRK3") scheme. Utilizing 1, 2, 4, 8, 16, 32 nodes of the JUWELS cluster module, during time integration, thirteen timings are respectively taken for sets of eRK3 iterations. The respective median is taken for the following graph depicting the strong parallel scalability. For



a single cluster node, there are about 22,500 elements per physical core, for sixteen nodes only about 1,400, i.e. an extreme degree of parallelism is considered. The plot shows that running CODA (version 2021.11) 2-level parallel (one process per socket, cf. above, “2Px24T”) is at least as fast as running it in classical MPI-only mode (“48Px1T”). In fact, hybrid is actually slightly faster than MPI-only. Both show a parallel efficiency of about 80% for 32 nodes (relative to a single node). Utilizing Hyperthreads with the hybrid parallel mode (“2Px48T”), a reduction of wallclock runtime by about 15% can be observed for lower numbers of cluster nodes. When utilizing more nodes, however, scalability breaks down at 16 nodes when utilizing Hyperthreads. The actual reason for this has not yet been identified. Load-balance issues may be involved in this particular case, but further investigations are needed.

The second HPC system considered is DLR’s CARA cluster. It is based on 1st generation AMD EPYC 7601 32-core processors. In contrast to the Intel architecture above, an AMD EPYC chip features multiple non-uniform memory-access (NUMA) regions, namely, for the CARA configuration, four UMA domains per chip, i.e. eight memory domains per cluster node. And in fact, it has turned out that running eight MPI processes per node, each exclusively bound to one memory domain, is the best setting for the considered case. Again the ONERA M6 case already considered above is used – but this time with a medium sized mesh with about 69 million elements (hexahedrons and prisms). For a single node, this results in approximately 1.08 million elements per physical core, i.e. very moderate parallelism. The 2-way symmetric multi-threading (SMT) is used here, i.e. 16 threads are run per “Core-Complex Die” (CCD). Each CCD consist of two Core Complexes (CXXs), each featuring an L3 cache for the four cores of an CXX. Thus, cache-coherence logic plays an important role. As the plot shows, utilizing 2-way SMT speeds up the simulation also for EPYC. Running CODA (version 2022.4) in hybrid-parallel mode is always faster than MPI-only, i.e. one single-threaded process per physical core. There is a noticeable reduction to 80% parallel efficiency when going from four to eight nodes.



(The reason has not yet been fully understood.) Increasing the number of nodes by a factor of 50, namely from eight to 400, parallel efficiency does not drop below 70%. This means that 25,600 physicall cores (running 51,200 threads) can be spent rather than only 512 cores (1024 threads) at an additional cost of only $80/70 - 1 \approx 14.3\%$. Putting it differently: The user is able to speed-up the simulation by a factor of almost 44 when using 400 nodes instead of just eight. This extreme scalability may allow running industrial sized cases over night, with only a minor additional charge. Note that 400 nodes correspond to about 1350 elements per thread (sub-domain) for the simulation setup considered. When using more than 400 nodes of CARA for this setup, fluctuations in the runtime measurements start to appear, indicating the limit of scalability – of the code in combination with the cluster – namely, other jobs affect the network performance. Finally, note that MPI-only (i.e. 64 processes per node) could only be run up to 3072 processes due to limitations in all-to-all communication during FSDM mesh partitioning.

6 Summary, Conclusion, and Outlook

The design of a hybrid parallelization of CFD on unstructured meshes, based on a 2-level domain decomposition, was presented as well as details of its implementation in FLIS, the infrastructure of CODA. The extreme scalability of CODA's parallelization for steady flows integrated by an explicit Runge-Kutta method was demonstrated for two current HPC clusters, one based on Intel processors, the other one on AMD processors.

This work shows that "MPI+X", i.e. a 2-level parallelization featuring "some" shared-memory level in addition to the process (distributed-memory) level (usually MPI), can allow for an enhanced utilization of massively parallel (homogenous) compute hardware. The "X" must be done right, though – and the same holds true for the "+", the coupling of the levels. CODA's 2-level domain decomposition for unstructured meshes seems on the right path in this regard.

Though the parallel scalability of explicit time integration may be further improved, the focus will be on implicit time integration. Here the solving of sparse linear systems needs to be pushed to the scalability limit – in combination with the non-linear parts of the solution process, resulting in interactions that need to be treated as parallel as possible just as well.

Acknowledgements: This work was supported through the Federal Aviation Research Programme in Germany ("LuFo"), grant 20X1704A. The authors thank the Jülich Supercomputing Centre (JSC) for providing computing time on the supercomputer JUWELS (cluster module).

REFERENCES

[1] T. Alrutz and et al. GASPI – a partitioned global address space programming interface. In R. Keller and et al., editors, *Facing the Multicore-Challenge III*, volume 7686 of *LNCS*, pages 135–136. Springer, 2013.

[2] R. Aubry and et al. Some useful strategies for unstructured edge-based solvers on shared memory machines. *Int'l J. Num. Methods in Engineering*, 85(5):537–561, 2011.

[3] R. Barrett and et al. MiniGhost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing. Technical Report 5294832, Sandia National Labs, USA, 2011.

[4] A. Basermann and et al. HICFD: Highly efficient implementation of CFD codes for HPC many-core architectures. In C. Bischof and et al., editors, *Competence in High Performance Computing 2010*, pages 1–13. Springer, 2011.

[5] J. M. Bull and et al. A microbenchmark suite for mixed-mode OpenMP/MPI. In *Evolving OpenMP in an Age of Extreme Parallelism*, pages 118–131. Springer, 2009.

[6] W. T. Gropp. MPI+MPI: Using MPI-3 shared memory as a multicore programming system. Presentation at SIAM Conf. Parallel Processing for Scientific Computing, Paris, April 2016.

[7] D. Grünewald and C. Simmendinger. The GASPI API specification and its implementation GPI 2.0. In *Proc. 7th Int'l Conf. PGAS Programming Models*. The University of Edinburgh, 2013.

[8] I. Hadade, F. Wang, M. Carnevale, and L. di Mare. Some useful optimisations for unstructured computational fluid dynamics codes on multicore and manycore architectures. *Computer Physics Communications*, 235:305–323, 2019. **open access**.

[9] J. Jägersküpper and C. Simmendinger. A novel shared-memory thread-pool implementation for hybrid parallel CFD solvers. In *Proc. Euro-Par 2011 Parallel Processing Conf.*, volume 6853 of *LNCS*, pages 182–193. Springer, 2011.

[10] N. Kroll and et al. DLR project Digital-X: towards virtual aircraft design and flight testing based on high-fidelity methods. *CEAS Aeronautical Journal*, 7(1):3–27, 2016.

[11] S. Langer, A. Schwöppe, and N. Kroll. The DLR flow solver TAU – status and recent algorithmic developments. In *52nd Aerospace Sciences Meeting, AIAA SciTech Forum*, volume 2014-0080, 2014.

[12] T. Leicht and et al. DLR-project Digital-X: Next generation CFD solver 'Flucs'. In *Deutscher Luft- und Raumfahrtkongress 2016*, 2016. http://www.dglr.de/publikationen/2017/420027.pdf.

[13] R. Löhner and J. D. Baum. Handling tens of thousands of cores with industrial/legacy codes: Approaches, implementation and timings. *Computers & Fluids*, 85:53–62, 2013.

[14] M. Meinel and G. Einarsson. The FlowSimulator framework for massively parallel CFD applications. In *Proc. Para 2010 – State of the Art in Scientific and Parallel Computing*. University of Iceland, Reykjavik, 2010.