2021 20th International Symposium on Parallel and Distributed Computing (ISPDC) | 978-1-6654-3281-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISPDC52870.2021.9521644

Distributed Domain Generation for Large-Scale Scientific Computing

1st Christoph Ertl
Chair of Computational Modeling and Simulation
Technical University of Munich
Munich, Germany
christoph.ertl@tum.de
orcid.org/0000-0002-3577-6365

2nd Ralf-Peter Mundani
Swiss Institute for Information Science
University of Applied Sciences of the Grisons
Chur, Switzerland
ralf-peter.mundani@fhgr.ch
orcid.org/0000-0001-6248-714X

***Abstract*—In this work, we present methods for distributed domain generation within the constraints of our decentral domain management concept. Here, all participating actors only have knowledge of their immediate neighbours, which are defined by geometric and hierarchical relations between nodes that represent subsets of the computational domain. We generate this domain following a hierarchical spacetree refinement. First, an initial tree is generated on every participating process. Second, this tree is distributed following a space-filling curve linearisation locally. Every process is assigned at least one leaf node of the initial tree, which acts as a starting point for the subsequent domain generation. From here, every process independently refines a subdomain using a decomposition method, which transforms a triangular surface-based geometry description into a volume-based one, using increasingly complex intersection tests. The resulting domain tree is distributed, yet neighbourhood references of neighbouring subtrees are not resolved. We combine the resolution of these relations with a 2:1 tree balancing, which involves the transfer of the surface of neighbouring subtrees. We provide results of a domain generation testcase, using an input geometry with 84,072 triangles on up to 896 processes of the CoolMUC-2 cluster segment of LRZ's Linux Cluster System. Here, we bring down the overall time it takes to generate an adaptively refined and balanced octree with depth d = 7 from 5.5 hours on one process to two seconds on 896 processes.**

***Index Terms*—Large-scale scientific computing, parallel algorithms, distributed algorithms, distributed systems, distributed domain generation, spacetrees**

I. Introduction

In the ever growing landscape of numerical computing on high-performance clusters, every part of a dedicated simulation codebase must be custom-tailored for the distributed hardware. With machines encompassing core counts in the range of millions of cores, every serial section of a code must be avoided as far as possible, to make efficient use of the parallel hardware. Within the codebase developed at our institute for the real-time simulation of physical phenomena described by the Navier-Stokes Equations [1], one of these limiting serial sections is the domain generation. It followed a centralised approach, where a single management instance is responsible for reading in input files containing geometrical information, generating the datastructure, partitioning the structure and distributing it to the responsible processes. Only then a computation in parallel is possible.

The central management instance has been identified as one of the main bottlenecks to performance and has been replaced recently. Our current approach employs a decentral approach to domain organisation, where the essential idea is to limit the domain view of each participating unit to their direct neighbours. The new approach affects various parts of the numerical code significantly. Among them, the octree-based domain generation, which had to be revamped from a central based one to support the decentral structure similarly. To this end, an algorithm has been devised, where all participating processes generate the computational domain up to a predetermined depth, before distributing the resulting leaf nodes of the geometry to be starting points for a subsequent local refinement. Furthermore, we introduce a balancing algorithm to only allow a maximum difference of the refinement depth of neighbouring tree nodes of one. Finally, we detail the resolution of the neighbourhood relations between the newly generated local trees of each process, both locally and across process borders.

The remainder of this article is structured in the following way. In the next section II, we detail all foundational concepts of our codebase. These include a short overview over different domain management concepts and the partitioning of our data structure. To conclude the section we introduce a spacetree generator to create a volume-based model from a surface-based (triangular) geometry description. The following section III introduces the distributed domain generation algorithms in detail, including the 2:1 tree balancing and the resolution of neighbourhood relations across different neighbouring processes. We show experimental results of the decentral approach in section IV before some conclusions and closing remarks in section V.

II. Foundations

Our fluid flow simulation framework is custom-tailored to be deployed on current top tier massive parallel machines. It has been tested on multiple high-performance clusters and showed good scaling behaviour on up to $140,000$ cores [2]. Nevertheless, we also identified bottlenecks, one of which

turned out to be the central management instance. In the following, we will give an overview over the different management concepts and detail briefly the old approach based on a global view of the domain and the current approach, which follows a local domain view. Furthermore, we detail another key component of the codebase, the partitioning scheme based on the idea of a spacetree refinement (with quadtrees as 2D and octrees as 3D representatives). We close the chapter with the description of the voxel based algorithm to generate our geometry description.

A. Domain management

In a parallel environment, the simulation domain must be partitioned and distributed to the responsible processes. When viewed as a graph, the domain partitions appear as nodes of the graph. Neighbouring partitions usually have to exchange bordering values and information during the simulation process. These data dependencies are represented by edges in the graph. The distribution of individual partitions, i.e. nodes, must satisfy two criteria, such that the computation is able to efficiently use the parallel hardware: a reasonably balanced distribution of individual partitions to processes and the minimisation of inter-process edges, minimising communication over the cluster network. Furthermore, if an adaptive mesh refinement and coarsening is to be employed, causing imbalances in partition size and therefore imbalances in computational load among the processes, a continuous load balancing is advantageous. All these tasks must be handled by a domain management instance.

There has been extensive effort into developing libraries that take the burden of domain management. The most well known are *metis* and its variant for distributed machines *parmetis* [3]–[8], which are based on various graph partitioning methods to find an efficient distribution of the simulation domain. And *p4est* [9]–[12], which is based on so called forests (distributed connected octrees) and space-filling curves to evaluate efficient distributions of these octrees to processes.

Domain management concepts can be classified by the domain view, that is the extent of knowledge of the domain graph, and the number of processes that hold this information. If the complete domain graph is available, one speaks of a global view. Consequently, a local view means the graph is only partially available. Both mentioned libraries employ a hybrid approach of the domain view. The partitioning of the domain is exclusive, each process has only a local view of its assigned partitions, however all processes share a macrostructure connecting the individual partitions. Changes to the overarching macrostructure are costly, but happen very rarely, such that these libraries work exceptionally well in practice and show scalability up to the maximum capabilities of current hardware.

At the initial design of our codebase, a domain management concept was chosen following a global view strategy on a single management instance [13]. This choice was motivated to support adaptive mesh refinement and coarsening during the runtime of the domain, allowing to dynamically rebalance the computational load. Having the complete domain graph available, state-of-art balancing techniques based on space-filling curves (SFC) can be employed. As mentioned above, employing this strategy has performed quite well on current hardware. Nevertheless, the requirement to communicate regularly with a single management instance has been identified as a bottleneck to performance, especially with ever growing numbers of available cores and therefore processes. An extension has been proposed to use multiple management instances, each responsible for a subset of the computational domain, while synchronising an overarching macrostructure among them, comparable to the approaches taken by *metis* and *p4est*. Yet, we believe, keeping any kind of metadata that has to be synchronised among all or a subset of processes, cannot be infinitely scaled.

To this end, we have developed a domain management concept based on an exclusive partitioning of the domain graph among the participating processes without any overarching structure that connects all subgraphs [14]. Each process only holds information about its own local partition and is aware of topological connections to neighbouring partitions. Information is exchanged only with processes holding connected partitions. Therefore, information travels through the domain, following a diffusion model, concentrically from its source. This information transfer is comparably slow when applied purely geometrical, but this is alleviated by making use of our hierarchical data structure, which allows information to travel through the hierarchy of the spacetree in addition to the geometrical neighbours. Nevertheless, global information exchange is intentionally avoided in our concept. Neither do we have to synchronise a macrostructure, nor do our management methods require any kind of global domain view. Most affected by this is the load-balancing after an adaptive mesh refinement and coarsening. Having only local domain information, we use a diffusion-based approach to exchange load between neighbouring processes. Limiting the number of diffusion steps certainly is unable to achieve a perfect balance for arbitrary large domains as a global view method would, however a theoretical scalability independent from the total size of the domain graph is only possible by limiting the domain view.

B. Partitioning

As mentioned above, we follow the idea of spacetrees to partition the simulation domain. Our graph starts with a root node, representing the complete domain at depth $d = 0$. The domain is then successively subdivided in each cardinal direction, generating subblocks in each refinement step. This subdivision is continued up to a predefined depth d_{max}. We also allow an adaptive refinement, in which blocks are only refined if they satisfy a defined condition. For example, if they contain geometry or other interesting features of the simulation. Since the refinement depth will be directly responsible for the resolution of the numerical mesh, a greater depth leads to a greater accuracy. An exemplary 2D data structure adaptively refined to depth $d = 5$ is illustrated in Figure 1.

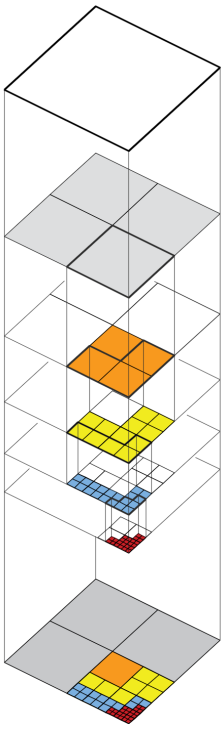


Fig. 1. A spacetree structure as a result of the partitioning of an arbitrary numerical domain: the refinements with all hierarchical levels are shown from top to bottom. The bottom depiction shows an agglomeration of the finest resolution available.

In addition to classical spacetrees, we allow varying subdivisions from one to seven in each cardinal direction, amounting up to 49 (2D) or 343 (3D) child nodes per refinement step. This allows us to represent complex domains with no degeneration of partitions by being overly stretched in one direction. A node is uniquely identified by a 64 bit integer, whereas the last nine bits are used to encode the position of a node in the parent's local coordinate system. With three bits per direction, we are left with eight representable positions. A recursive subdivision of more than five is physically not feasible [1], therefore, we use seven valid positions and one invalid position.

In our implementation, each graph node is then discretised by a regular, orthogonal, block-structured grid. Each grid comprises cells of size $s_x \times s_y \times s_z$, which store all necessary problem variables such as velocities, pressure or temperature values. Furthermore, each grid is surrounded by a halo of ghost cells storing a current copy of the neighbouring cell values. Since each node is discretised by a grid, our data structure represents the simulation domain in various resolutions throughout the whole spacetree hierarchy.

Each node has two types of neighbours, hierarchical neighbours and geometrical neighbours. Hierarchical neighbours are parent or child nodes from the spacetree refinement. They all represent the same space or parts thereof in different resolutions. Geometrical neighbours are the "real" neighbours, located adjacent to a node in the simulation space. Here, we only consider geometrical neighbours on the same refinement depth. These neighbours discretise the space with the same resolution. Figure 2 illustrates all hierarchical and geometrical neighbours of a node (orange checkers) in two dimensions. A node has exactly one parent (grey on top level). The number of child nodes depends on the subdivision chosen. Here, a bisection in each cardinal direction per refinement step was used, which amounts to four children (yellow on bottom level) per refined node. In two dimensions, a regular grid has at most four geometrical neighbours on the same refinement level (solid orange). Nodes at the domain border have fewer. If the domain is non-uniformly refined, nodes may also have fewer geometrical neighbours. In accordance with the premise of minimising the connections among processes and avoiding broadcast operations, each process only communicates with other processes that were assigned neighbouring nodes.

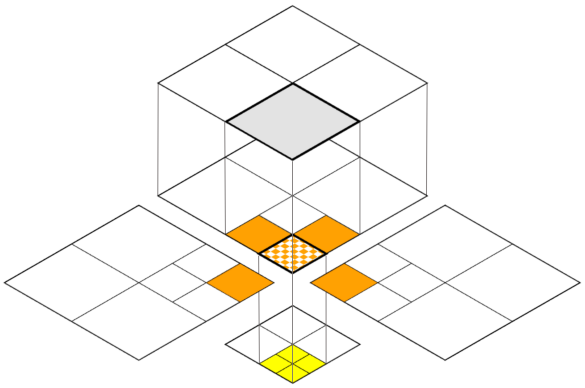


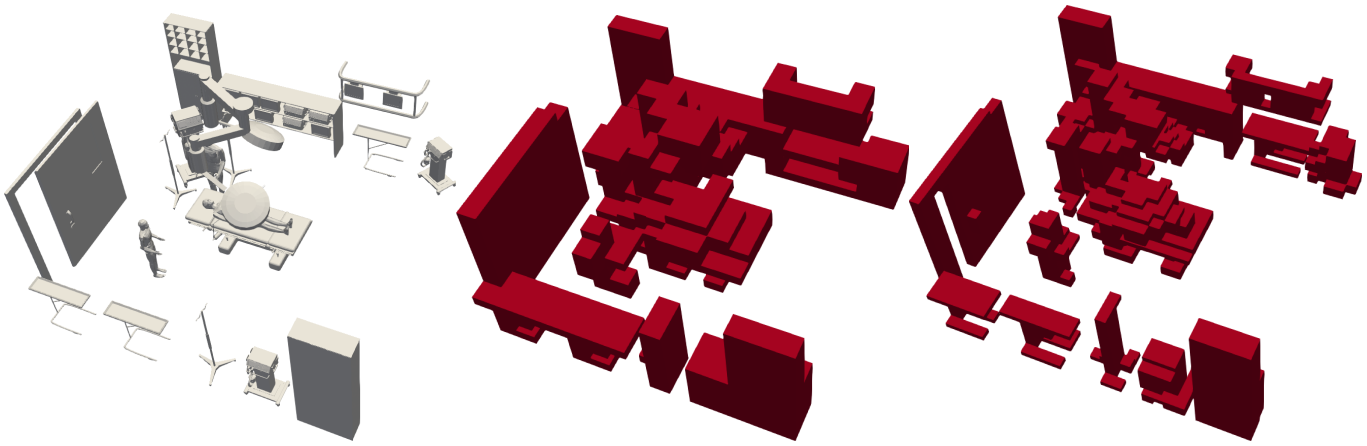
Fig. 2. An illustration of all geometrical and hierarchical neighbours of a node in two dimensions. When a node is refined, the space is bisected in each cardinal direction, spawning four child nodes.

C. Geometry generation

In order to set up simulation scenarios involving complex geometries, a fast and reliable way is needed to generate a volume-based description according to our data structure. Even more so, if scenarios should be evaluated in which geometries are moving or the mesh is dynamically refined during runtime, which requires a re-evaluation on the fly. References [15], [16] describe the idea of a spacetree generator to create a volume-based model from a surface-based description.

The general idea is again following a spacetree refinement, this time however in an adaptive fashion. In three dimensions the geometrical primitive is a voxel. If a voxel is refined, each dimension is subdivided, resulting in a number of child voxels. The decision if a voxel should be refined is based on a geometrical intersection test with the surface-based description. If one triangle is inside, touches or intersects a voxel, the voxel is added to the candidate list for refinement. This process is carried out starting from a single voxel containing the complete domain and is subsequently executed for each successive refinement depth, until a prescribed depth has been reached, or no voxels are flagged as refinement candidates during the intersection tests anymore. Reference [16] proposes a fast generator based on intersection tests with increasing costs. The logic behind this is, while applying less complex, inexpensive tests, many candidates can be discarded quickly before more expensive tests have to be applied. Figure 3 shows an example from [17] and [18], an operating theatre located at the university hospital "Klinikum rechts der Isar" in Munich. We used the surface model to generate a volume description using the voxel generator. Figure 3a shows the input geometry, while figure 3b and figure 3c depict a volume based model refined up to depths $d = 5$ and $d = 6$. In each cardinal direction, a bisection was used when subdividing voxels, following the classical octree scheme.

The voxel generator has been adapted to our data structure in [19]. A first pass of the algorithm generates the octree, which turns into our domain graph. Subsequently each node



(a) Surface-based model (b) Volume-based model, depth $d = 5$ (c) Volume-based model, depth $d = 6$

Fig. 3. Generation of the volume-based models from a surface-based description, resolved up to depths $d = 5$ and $d = 6$.

is discretised with a computational mesh that consists of a predefined amount of cells. Then, a second run is carried out. Here intersection tests are performed on the actual cells to set the geometrical boundary conditions directly.

Until lately, we used this algorithm on the central management instance to generate the domain and set the boundary conditions on the created cells. The domain was then distributed using a Z-Order SFC to all participating processes. Not only the distribution of the domain from a single source was identified as a bottleneck, but also the memory requirement of this single instance has proven to be prohibitive for large-scale examples from real-world applications. With the recent overhaul of the management concept of our codebase, the domain generation had to be revamped as well, following a decentral approach.

III. Distributed domain generation

With the abolition of our central domain management instance, there is still the possibility of using the old domain generation facilities by selecting a single process from the cluster and performing the initial generation and distribution process from this single source as described before. This naturally does not solve any of the bottlenecks, therefore we updated the domain generation to make use of the distributed hardware.

The basic idea is to have every process be responsible for the generation of its own share of the simulation domain. Therefore, every process needs a starting voxel to carry out a spacetree refinement. As such, the actual generation is a multistage process. In the first stage, each process either uniformly or adaptively generates a domain following a uniform spacetree refinement or an adaptive approach, using the voxel generator with a triangular-based input file. The key aspects of this initial domain structure are a refinement up to a depth, in which the amount of leaf nodes of the generated domain tree are at least equal to the amount of participating processes. Furthermore, only the tree nodes are generated without initialising the corresponding computational grids, which keeps the memory requirement low. Figure 4a illustrates an example domain uniformly refined in a simplified fashion. Four participating processes generate the initial domain up to a depth of three, with four leaf nodes depicted in green on the deepest refinement level.

At this point, every process has the complete domain information available locally. This allows to resolve the neighbourhood relations on all nodes directly. Metadata about each nodes' parent and child nodes are stored with the node during generation of the tree. In other words every node is able to reference its parent and possible children, making it possible to traverse the complete domain tree and find horizontal neighbours easily. Furthermore, the global view of the domain allows the computation of a space-filling curve. Subsequently, each process computes the final distribution according to the linearisation from the SFC and updates all local metadata accordingly. After all nodes that do not belong to a process are deleted, we arrive at an initial domain configuration exactly equal to a configuration we would have got with a central management instance. Additionally, no communication has taken place so far. Figure 4b shows a simplified illustration of the current domain distribution after the deletion of all foreign nodes. The now deleted nodes on a process are signified with a hatching.

Each process has now been assigned an exclusive partition of the initial tree with at least one of its leaf nodes. In the third stage, each process individually generates a new subtree with its leaf nodes as a starting point following the geometry generation method outlined in section II-C. Figure 4c again shows a simplified view of the current configuration in which each process has refined its local subtree at the bottom of the illustration. For the local subtree, we are again able to resolve the neighbourhood relations by traversing the tree using the hierarchical domain structure. However, neighbouring trees are not available, so inter-process neighbourhood cannot be resolved locally. One remedy would be to keep the domain

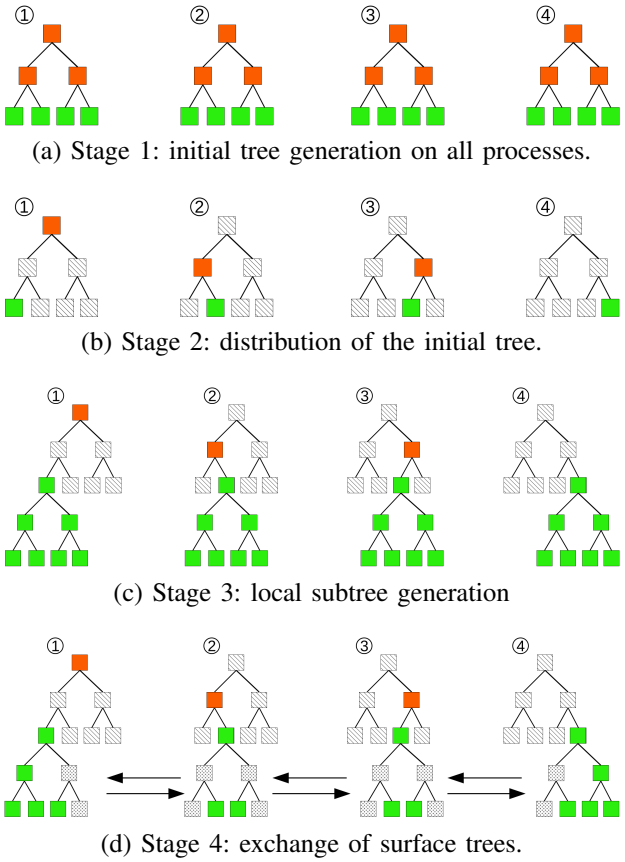
generation purely local, that means every process generates the subtrees of all geometrical neighbours to its leaf nodes in addition to its own local subtree. Having the neighbouring subtrees available locally in turn allows to resolve the neighbourhood information without any communication. It is not necessary to update neighbourhood metadata of the foreign trees. Furthermore, after the metadata is updated, they may simply be deleted.

Our code supports ghost layer updates across different refinement depths. Nevertheless, numerical constraints limit the amount of depth discrepancies between neighbouring nodes. Our implementation therefore balances the ensuing local subtrees following a 2:1 balance constraint. In other words, geometrical neighbouring nodes cannot exceed a depth difference of more than one. This inevitably requires communication between processes holding neighbouring subtrees, therefore, we are able to combine communication of the neighbourhood metadata with tree information that allows us to balance the trees. This means, we do not generate foreign subtrees to resolve neighbourhood information, but rather use the communicated metadata to combine balancing and updating neighbourhood references. In the next section, we illustrate the details of this balancing and the neighbourhood resolution.

A. 2:1 tree balancing and resolution of inter-process neighbourhood relations

Refining a node without also refining its neighbours leads to a discrepancy in refinement depth of the space discretised. Consequently, the grids that discretise geometrical neighbouring nodes are non-conforming and the problem of hanging nodes arises. As mentioned above, our data structure allows an arbitrary number of hanging nodes, using the space tree hierarchy to exchange the necessary ghost cell data through interpolation from finer resolutions or prolongation from coarser ones. Yet, to ensure the convergence of numerical solvers by limiting the interpolation errors, we require a difference between neighbouring refinement levels of at most one level. This problem is well known in literature as the tree balancing problem and has been extensively studied for octrees, for example in [20], [21] and [22]. Again, our completely local domain view does not allow the use of algorithms requiring any kind of global knowledge of the domain structure. Therefore, the starting point of our investigation is a local 2:1 tree balancing algorithm.

Prerequisite for the local tree balancing is a complete local tree structure, where every node is connected and can be reached. The pseudo code for the method is illustrated in algorithm 1 and its main task is to determine whether the level of a geometrically neighbouring node is two levels finer than the current node that is treated. If that is the case, the node needs to be refined. The method therefore loops over all leaf nodes in the tree and performs a number of checks. Every other node is already refined and does not need to be taken into account. First, all geometrical neighbours of the node are referenced. For any refined neighbours, their children



(a) Stage 1: initial tree generation on all processes.

(b) Stage 2: distribution of the initial tree.

(c) Stage 3: local subtree generation

(d) Stage 4: exchange of surface trees.

Fig. 4. Four stages of the distributed tree generation algorithm: generation of the initial tree on all processes, subsequent deletion of all foreign nodes and generation of individual subtrees with the previous leaf nodes as starting points. The last stage depicts the exchange of the surface trees to balance local subtrees and update neighbourhood metadata.

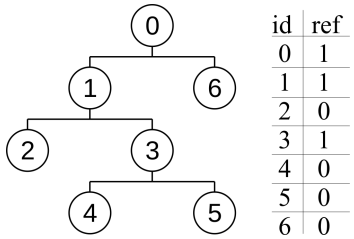
are referenced. Finally, the children are checked if they lie bordering to the original node and whether they are refined themselves. If both requirements are true, we have determined that the discrepancy between the refinement depth is at least two or greater and the original node has to be refined. Any remaining checks for the current node can be forfeited at this point. Tree balancing is repeated until a complete iteration through all leaf nodes does not yield a single refinement any more. A simple flag, set when a node is refined and unset between iterations of the method, is sufficient for this task.

After locally balancing the subtrees, information has to be exchanged between processes that hold neighbouring subtrees, i.e. subtrees generated from geometrically neighbouring intermediate leaf nodes. It is sufficient to exchange information concerning the surface in the direction of the neighbouring subtree. The surface of a spacetree is a tree itself, only with one dimension less. There are multiple ways to store spacetrees very memory efficiently. One is converting the tree structure into a depth first list of bits. Each bit encodes whether a node is refined or not. With the knowledge of the subdivision spacing, i.e. the amount of new nodes generated when a refinement happens, the tree can be rebuilt from this list of bits. Figure 5 shows an example spacetree with a subdivision of two for

Algorithm 1 Local tree balancing method.

```
for all leaf nodes do
  for all geometrical neighbours do
    if neighbour is refined then
      for all child nodes do
        if children are bordering the original node and
        are refined then
          refine original node
          jump to next node
        end if
      end for
    end if
  end for
end for
```

every refinement. The list is generated going through the tree from top to bottom first and from left to right second. The *id* is an increasing number, signifying the order of traversal of the tree. *ref* takes the value 1 if the node is refined and 0 otherwise. To transfer surface information of surrounding subtrees, we send this bit encoding of the surface tree, followed by a list of identifiers to uniquely reference the nodes making up the surface in the computational domain. In figure 4d the transfer of the subtree surfaces is illustrated in a simplified fashion. The nodes that make up the surface to be transfered are depicted with a dotted pattern.



id	ref
0	1
1	1
2	0
3	1
4	0
5	0
6	0

Fig. 5. Depth-first encoding of an example spacetree (binary-tree) with seven nodes in total. *id* refers to a consecutive numbering of nodes following their traversal, *ref* is encoding the tree structure and encodes whether a node identified by *id* is refined (1) or not (0).

Both lists allow the receiver process to generate a mock tree, which can be traversed similarly as the processes' own subtree. Furthermore, it contains all neighbourhood references of the foreign nodes, which are used to update the references of the locally held nodes. Having all the neighbourhood surface trees available, after updating the neighbourhood references of all locally held nodes, the local tree balancing can be applied similarly as before. Newly generated nodes on one process may warrant subsequent updates on remote processes and vice versa. As such, the cycle of communicating the surface information, updating neighbourhood references of nodes that have gotten new neighbours and balancing the local tree with the help of the surrounding mock trees, has to be repeated multiple times. After the first cycle though, only changes to the surface structure have to be communicated and not the complete subtree surface. After all cycles are completed, the mock trees are deleted. The upper bound for the amount of needed cycles is given by the maximum refinement depth minus the depth of the initial tree minus one (since we allow an imbalance of one between neighbouring nodes). Even if subsequent cycles yield no more refinements anymore, we cannot break early. Refinements caused by nodes on the opposite side of a remote subtree may only be visible after refinements have cascaded through the remote subtree in a later cycle.

There are two remarks worth mentioning. First, nodes generated from the 2:1 balancing cannot have geometry boundary conditions. As such, further intersection tests on the cells of the new nodes are not necessary. If a node had parts of the geometry in the first place, it would have been refined previously when we generated the subtree. Second, it is not possible to employ the same strategy of generating the subtrees from neighbouring intermediate leafs on a process and use these trees to apply a local 2:1 tree balancing. A node is identified by the rank of the process it is held by and a unique id, local to the owning process. This id cannot be computed by a foreign process since the exclusive domain distribution prohibits knowledge about neighbours of neighbours (second level neighbours). If nodes are refined due to an inter-process tree balancing, these refinements cannot be seen by the neighbouring subtree on the other side. With this we conclude the section explaining our decentral domain generation within the constraints given by the data structure and the domain management concept.

IV. Implementation results

In order to show the viability of the decentral domain generation, we used the operating theatre from section II-C, illustrated in figure 3 again, an example we have previously computed using our old centralised approach. The underlying geometry has dimensions of $6.3m \times 6.25m \times 3.5m$, the input file contains $84,072$ triangles. All tests were run on the Linux Cluster System provided by the Leibniz Supercomputing Centre (LRZ) [23]. We used the CoolMUC-2 cluster segment [24], which consists of 812 28-way Intel Xeon E5-2690 v3 Haswell-EP nodes with Infiniband FDR14 interconnect and two hardware threads per physical core. It's theoretical peak performance amounts to 1,400 TFlop/s. The used compiler was the Intel Compiler in version 19.0.

We used our distributed domain generation method to generate and balance the domain with refinement depths $d = 3$ to $d = 7$. We used increasing process counts starting from 1 process on 1 cluster-node up to 896 processes, with 28 processes per cluster-node and 32 nodes. The chosen subdivision $r_x \times r_y \times r_z$ is $2 \times 2 \times 2$, resulting in 8 child nodes for every refined node (the classic octree). Each node is discretised after the initial partitioning with a grid of $8 \times 8 \times 8$ cells. Each individual test combination was run multiple times to calculate the truncated mean, where we discarded extreme outliers. The Linux Cluster is a shared resource that is used by many researchers simultaneously. Therefore, outliers are caused by other running applications that also use the interconnect.

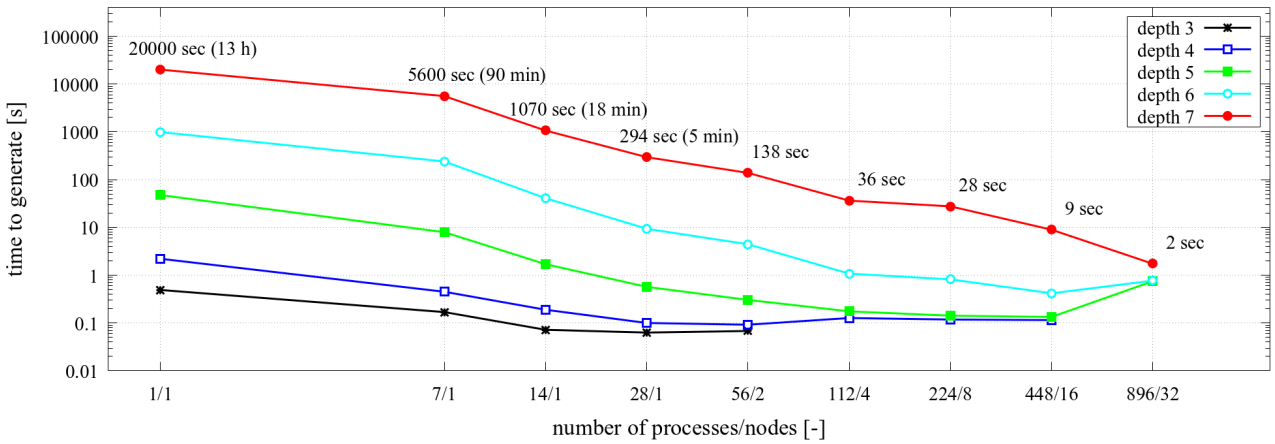
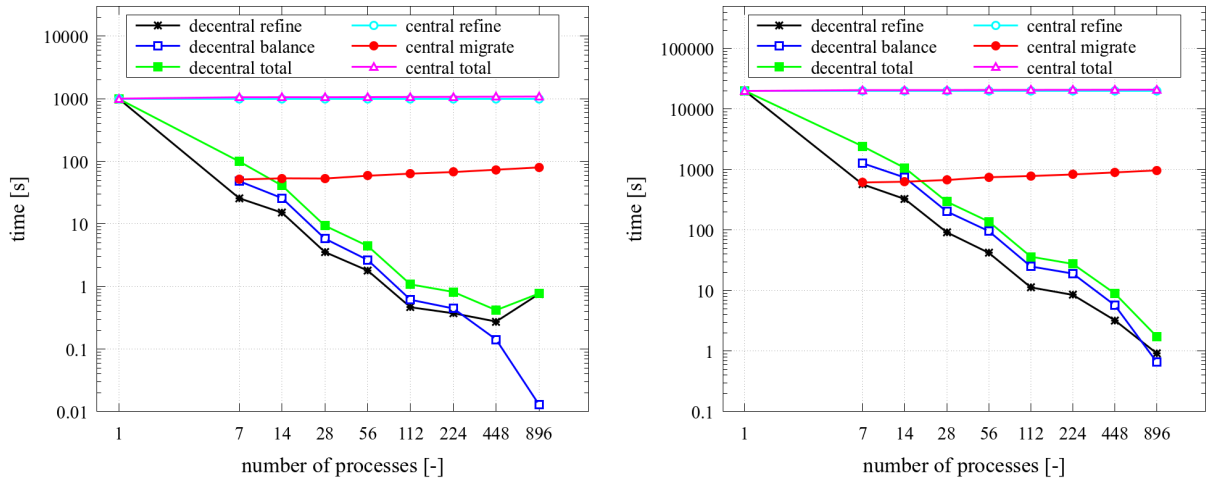


Fig. 6. Times for domain generation including a 2:1 tree balancing and resolving the neighbourhood relations for different domain resolutions from refinement depths from $d = 3$ to $d = 7$, measured against the amount of processes used from 1 to 896.



(a) Domain adaptively refined to depth $d = 6$

(b) Domain adaptively refined to depth $d = 7$

Fig. 7. Times for domain generation using the described decentralised methods compared against the former centralised methods. The measurements are broken down into a refinement step to generate the domain tree and a balance step for the decentral approach and a migration step for the central approach. The times are measured against the amount of processes used from 1 to 896.

Consequently, without the outliers, the given truncated mean agrees very closely with the minimum observed times. There are no measurements for refinements of depth $d = 3$ and processor counts from 112 onwards. The intermediate tree needed such that every participating process is assigned at least one leaf grid must be refined to depth $d = 3$. Therefore, every process performs the complete refinement locally without any inter-process communication and deletes all foreign nodes according to the SFC linearisation. The same is true for refinements of depth $d = 4$ and a processor count of 896.

Figure 6 illustrates the measurements. We observe a steady decline for generation times with increasing processor counts for a refinement depth $d = 7$. Generating the domain takes approximately $20,000$ seconds or roughly to 5.5 hours on a single process. Increasing the processor count, the time comes down to approximately two seconds using 896 processes eventually. For comparison, we posed the former centralised method for domain generation against the new decentral approach. Figure 7a and figure 7b illustrate these measurements. The times shown for the decentral domain generation are further broken down into the adaptive generation of the domain tree and the migration from the central instance to all participating processes. The times for the 2:1 balancing were omitted, because they have no significant influence on the total time. The time for the load-balancing is included in the migration, where nodes are sorted according to their Morton ordering and equal chunks of nodes are distributed to their respective processes. The sorting itself has similarly no significant influence on the time, though. Finally, the total time as sum of the former two is depicted. The new approach is similarly broken down. The first part is the generation of the domain tree, made up by the generation of the intermediate

tree plus the refinement of the local leaf nodes. A migration step is not needed here, as all nodes are generated on their respective processes. The 2:1 balancing however, requires much more effort and communication over the network and is significant. Again, the total time as sum of the two parts is illustrated likewise.

The central approach always takes the same amount of time with around 15 minutes for depth $d = 6$ and around 5.5 hours for depth $d = 7$ to generate the domain tree, as no parallelisation is used. The time it takes to migrate the nodes to their respective processes is slowly increasing the more processes are involved, from around 50 seconds at 7 processes to 80 seconds at 896 processes at depth $d = 6$. For depth $d = 7$, the times increase from 613 seconds to 971 seconds. For the measured range of processes used, the generation is clearly the dominating factor. Both parts of the decentral approach, the tree generation and the balancing clearly benefit from the parallelisation. We observe a steady decline for generation times with increasing process counts for both refinement depths. The single process performance is equal to that of the centralised approach. Increasing the process count, the time comes down to approximately 2 seconds using 896 processes and a depth of $d = 7$. For depth $d = 6$ we observe a similar benefit from using more processes. However, at 896 processes, the time for the refinement stage increases again. At 448 the intermediate tree is refined on every process up to depth $d = 3$ resulting in an average number of leafs per process of 1.1. For 896 processes we need to refine the intermediate tree one level further. Using the classical octree with 8 children, the average number of processes comes out to 4.6. This explains the added time, each process has to refine the intermediate tree one level further, refining also non local nodes. Moreover, the amount of individual subtrees to generate is increased as well.

The speedups we observe for depths $d = 6$ and $d = 7$ are on average four times the amount of processes used. We mainly attribute this superlinear speedup to cache effects. Storing the complete domain tree on a single node far exceeds the size of faster caches. Increasing parallelisation not only shares workload, but also splitting up the domain tree allows for subtrees to fit into cache memory and allows faster access. The speedup in the balancing stage has multiple sources. The intermediate tree is balanced on all processes locally. This happens only once and is comparably very cheap. Increasing the process count leads in general to less subtrees to balance. In the best case, decreasing the amount of subtrees by one saves communication with four neighbours (3D). Furthermore, with the intermediate tree refined deeper, the subtrees are refined less, decreasing the effort to balance the local subtrees and the surface areas to communicate.

In figure 8 we show memory measurements during the domain generation process. The memory of the intermediate tree is increasing incrementally with the height of the tree. The intermediate tree for 896 processes refined to depth $d = 4$ with $4,681$ nodes in total requires roughly 1.9 MB. After the intermediate tree has been generated, nodes that belong

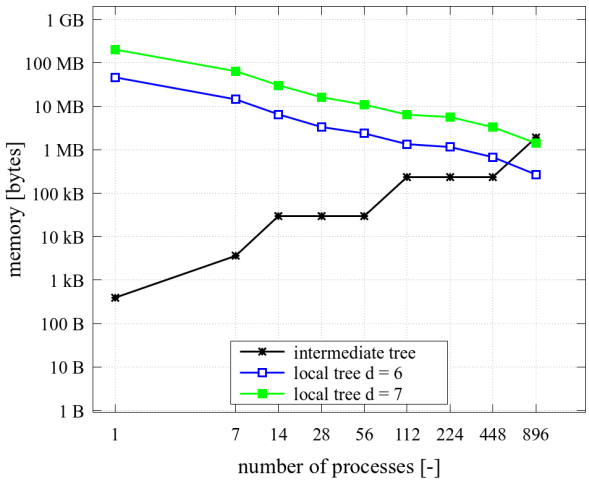


Fig. 8. Memory requirement of the intermediate tree and the complete node share of the process with the most nodes for depths $d = 6$ and $d = 7$ measured against process counts from 1 to 896.

to other processes are deleted, leaving a process with only its assigned leaf nodes and their share of the tree with an impact of around 2 kB. The memory requirement for the complete local node share decreases with the amount of processes used. The complete tree refined to depth $d = 6$ requires 45 MB and 200 MB for depth $d = 7$. The process with the largest share of the completely refined tree to depth $d = 6$ bears 270 kB and 1.4 MB when refined to depth $d = 7$. At 896 processes used, we observe that the memory requirement of the intermediate tree becomes the determining factor in how much memory must be available per process. At this point, using more processes will increase memory consumption again, up to the point were the intermediate tree and the final tree match each other at approximately half a million used processes for depth $d = 7$. At which point, the number of nodes on processes is very low and a simulation run won't be resource efficient. Nevertheless, the overall memory consumption has been lowered in the observed range of processes by two orders of magnitude compared to the centralised approach and can even in the worst case, never be higher than before.

V. Conclusions

From our measurements, we conclude two main findings. Our decentral domain generation succeeds in bringing the domain generation times down multiple orders of magnitude, within the constraints given by our domain management concept. A simulation preprocessing that took 5.5 hours could be brought down to two seconds in one case. Furthermore, the memory requirement on a single process has been lowered, in all observed cases, from being able to store the complete domain tree including all discretised grids to either only the local subtree of a process plus its share of the initial tree and the corresponding discretised grids or the intermediate tree. Configurations where the impact of the intermediate tree's memory requirement outweigh the requirement of the local

node share, are deemed unfit from an efficiency standpoint, though.

One caveat that has to be mentioned, is that the ensuing domain configuration is not yet load balanced. Only the intermediate tree is distributed following an SFC linearisation. Depending on the geometry, the resulting domain might be highly imbalanced when it comes to the number of nodes per process. Furthermore, the local view of the domain prohibits further use of an SFC linearisation to rebalance the domain. One possible remedy, would be to have an intermediate actor that after the domain has been generated and balanced, gathers the domain metadata, computes a balanced distribution and orders all processes to transfer nodes to their designated targets. Since this operation has to be carried out only once, the added cost could be justified. Another approach, we illustrated in [14], is a diffusion based load-balancing. This balancing is carried out continuously during runtime of the simulation to support adaptive mesh refinement and coarsening. It could similarly be used to achieve a balanced state after the domain generation.

Another valid criticism is, even though the decentral domain management promises a theoretical infinite scalability, the domain generation does not. To arrive at the point at which every participating process is assigned at least one leaf node, the intermediate tree has to be refined down to a depth on where at least as many leaf nodes as processes exist. The depth needed increases with the amount of processes participating. A conclusive solution to the increasing sequential part has yet to be found.

In conclusion, this article illustrates a distributed octree-based domain generation under the constraint of a decentral approach to domain organisation, where the essential idea is to limit the domain view of each participating unit to their direct neighbours. To this end, an algorithm has been devised which refines an input geometry on all processes up to a predetermined depth, before distributing the resulting leaf nodes of the geometry to be starting points for a subsequent refinement on their respective processes. We have discussed the key points of the approach, including the distribution of the initial tree, avoiding costly communication for the neighbourhood resolution by processes refining the complete initial tree and a combined neighbourhood resolution and tree balancing for the generated subtrees.

ACKNOWLEDGMENT

The authors gratefully acknowledge the Leibniz Supercomputing Centre (LRZ) for funding this project by providing computing time on its Linux-Cluster. Without their kind support, parts of this work would not have been possible.

REFERENCES

[1] J. Frisch, "Towards massive parallel fluid flow simulations in computational engineering," Ph.D. dissertation, Technische Universität München, 2014.

[2] J. Frisch and R.-P. Mundani, "Measuring and comparing the scaling behaviour of a high-performance CFD code on different supercomputing infrastructures," in *17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2015, pp. 371–378.

[3] G. Karypis, "METIS - Serial graph partitioning and fill-reducing matrix ordering," 2013. [Online]. Available: http://glaros.dtc.umn.edu/gkhome/views/metis

[4] G. Karypis and K. Schloegel, "ParMETIS - Parallel graph partitioning and fill-reducing matrix ordering," 2013. [Online]. Available: http://glaros.dtc.umn.edu/gkhome/views/metis

[5] D. LaSalle and G. Karypis, "Multi-threaded graph partitioning," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 225–236.

[6] K. Schloegel, G. Karypis, and V. Kumar, "Parallel static and dynamic multi-constraint graph partitioning," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 3, pp. 219–240, 2002.

[7] K. Schloegel, G. Karypis, and V. Kumar, "A unified algorithm for load-balancing adaptive scientific simulations," in *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE, 2000, pp. 59–59.

[8] G. Karypis and V. Kumar, "Parallel multilevel graph partitioning," in *Proceedings of International Conference on Parallel Processing*. IEEE, 1996, pp. 314–319.

[9] C. Burstedde, L. C. Wilcox, and T. Isaac, "p4est: Parallel AMR on forests of octrees," 2019. [Online]. Available: https://www.p4est.org/

[10] C. Burstedde, L. C. Wilcox, and O. Ghattas, "p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees," *SIAM Journal on Scientific Computing*, vol. 33, no. 3, pp. 1103–1133, 2011.

[11] T. Isaac, C. Burstedde, L. C. Wilcox, and O. Ghattas, "Recursive algorithms for distributed forests of octrees," *SIAM Journal on Scientific Computing*, vol. 37, no. 5, pp. C497–C531, 2015.

[12] C. Burstedde, "Parallel tree algorithms for AMR and non-standard data access," *ACM Transactions on Mathematical Software (TOMS)*, vol. 46, no. 4, pp. 1–31, 2020.

[13] J. Frisch, R.-P. Mundani, and E. Rank, "Resolving neighbourhood relations in a parallel fluid dynamic solver," in *2012 11th International Symposium on Parallel and Distributed Computing*. IEEE, 2012, pp. 267–273.

[14] C. Ertl and R.-P. Mundani, "Domain management and adaptive mesh refinement with minimal information," *Manuscript submitted for publication*, 2021.

[15] H. Samet, *The design and analysis of spatial data structures*. Addison-Wesley Reading, MA, 1990, vol. 85.

[16] R.-P. Mundani, *Hierarchische Geometriemodelle zur Einbettung verteilter Simulationsaufgaben [Hierarchical Geometry Models to Embed Distributed Simulation Tasks]*. Shaker, 2006.

[17] P. Wenisch, "Computational steering of CFD simulations on teraflop-supercomputers," Ph.D. dissertation, Technische Universität München, 2008.

[18] M. Pfaffinger, "Interaktive strömungssimulation auf hochleistungsrechnern unter anwendung der lattice-boltzmann methode [interactive fluid flow simulation on high-performance computers using the lattice-boltzmann method," Ph.D. dissertation, Technische Universität München, 2012.

[19] J. Frisch and R.-P. Mundani, "Multiskalen-strömungssimulation eines kraftwerkskomplexes auf höchstleistungsrechnern [Multiscale fluid flow simulation of a power plant on high-performance computers]," in *Proceedings des 22. Forum Bauinformatik*, 2010.

[20] T. Isaac, C. Burstedde, and O. Ghattas, "Low-cost parallel algorithms for 2:1 octree balance," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 426–437.

[21] D. Malhotra and G. Biros, "A distributed memory fast multipole method for volume potentials," *Institute for Computational Engineering and Science, University of Texas at Austin. padas. ices. utexas. edu/static/papers/pvfmm. pdf*, 2014.

[22] H. Sundar, R. S. Sampath, and G. Biros, "Bottom-up construction and 2:1 balance refinement of linear octrees in parallel," *SIAM Journal on Scientific Computing*, vol. 30, no. 5, pp. 2675–2708, 2008.

[23] "LRZ Linux Cluster dokumentation," 2021. [Online]. Available: https://doku.lrz.de/display/PUBLIC/Linux+Cluster

[24] "LRZ CoolMUC-2 dokumentation," 2021. [Online]. Available: https://doku.lrz.de/display/PUBLIC/CoolMUC-2